Recruta Social:
A concepção e o desenvolvimento de um serious game para a promoção do controle social

João Lucas Fabião Amorim

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

João Pessoa, 2019

João Lucas Fabião Amorim

Recruta Social

Monografia apresentada ao curso Ciência da Computação do Centro de Informática, da Universidade Federal da Paraíba, como requisito para a obtenção do grau de Bacharel em Ciência da Computação

Orientadora: Profª. Drª. Danielle Rousy Dias da Silva

Maio de 2019

Catalogação na publicação
Seção de Catalogação e Classificação

A524r Amorim, Joao Lucas Fabiao.
Recruta Social: A concepção e o desenvolvimento de um serious game para a promoção do controle social / Joao Lucas Fabiao Amorim. - João Pessoa, 2019.
162 f. : il.

Orientação: Danielle Silva.
Monografia (Graduação) - UFPB/CI.

1. Jogos sérios. 2. Design de jogo. 3. Desenvolvimento de jogo. 4. Unity. 5. Controle social. I. Silva, Danielle. II. Título.

UFPB/CI

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

Trabalho de Conclusão de Curso de Ciência da Computação intitulado **Recruta Social** de autoria de **João Lucas Fabião Amorim**, aprovada pela banca examinadora constituída pelos seguintes professores:

Prof. Dr. Danielle Rousy Dias da Silva
UFPB

Prof. Dr. Gustavo Henrique Matos Bezerra Motta
UFPB

Prof. Dr. Eudisley Gomes dos Anjos
UFPB

Coordenador(a) do Departamento Centro de Informática
Gustavo Henrique Matos Bezerra Motta
CI/UFPB

João Pessoa, 16 de maio de 2019

Centro de Informática, Universidade Federal da Paraíba
Rua dos Escoteiros, Mangabeira VII, João Pessoa, Paraíba, Brasil CEP: 58058-600
Fone: +55 (83) 3216 7093 / Fax: +55 (83) 3216 7117

“Educação não transforma o mundo.
Educação muda as pessoas.
Pessoas transformam o mundo.”
Paulo Freire

DEDICATÓRIA

Dedico este trabalho a todas as pessoas envolvidas no projeto “Recruta Social” e que exerceram um papel importante para que atingíssemos os nossos objetivos. Além disso, dedico também para todos os que tenham interesse e entusiasmo em fazer parte dessa área de desenvolvimento de jogos.

AGRADECIMENTOS

Os meus sinceros agradecimentos vão primeiramente a Deus por me tornar capaz de superar os desafios, por ter me ensinado a ter compaixão e ajudar o próximo, além ser a minha fortaleza nos momentos mais difíceis. Agradeço muito a minha família, pela educação e valores aprendidos, pela minha edificação e fortalecimento do meu caráter. Quero dar um agradecimento bastante especial para as minhas avós que sempre cuidaram de mim, sem elas, não são o que eu estaria fazendo da minha vida.

Meus próximos agradecimentos vão para os meus pais, por terem me criado e me educado, além de tornarem o meu futuro em uma realidade e serem o alicerce da minha vida. Além disso, quero agradecer a todos os meus irmãos, que são o meu porto seguro nos momentos mais desafiadores e por trazerem tanta alegria na minha vida. Também quero agradecer a Alessa pelos ensinamentos, que vão desde do fundamental até os dias de hoje, e pela ajuda na revisão ortográfica deste trabalho, e a Gabriela Duarte, minha namorada, pelo apoio que ela me deu nesses momentos finais da minha graduação, por ser uma pessoa muito especial para mim, por compartilhar amor e carinho, e ter bastante confiança e paciência comigo.

Meus sinceros agradecimentos vão também para a professora Danielle Rousy, por ter guiado e orientado, não apenas este trabalho, mas também o projeto “Recruta Social”, por ter me dado a oportunidade de trabalhar nesse projeto, que me trouxe muita experiência e aprendizado. Não posso deixar de agradecer também aos meus amigos, em especial os que eu adquiri durante o meu tempo na universidade, por terem compartilhado vários momentos e desafios, proporcionados pela graduação. E por fim, quero agradecer a todos os professores pelos quais eu passei durante a graduação, pelos conteúdos ministrados e pelos ensinamentos que serão a base para o meu futuro e a minha formação como profissional.

RESUMO

O enfoque do trabalho aqui apresentado, considera o crescente mercado de jogos, que atualmente estabelece relação com o mundo do entretenimento, concomitante ao universo *dos jogos sérios,* realidade cada vez mais significativa na evolução de uma sociedade mais consciente e capaz de discernir sobre valores e *controle social*, contexto que se consolida a cada dia. Para tanto, mostrou-se uma visão que contemplou o trabalho de *design* e desenvolvimento do jogo ”Recruta social”, projeto idealizado por um grupo de educandos da Universidade Federal da Paraíba em parceria com o TCE-PB e oportunizou aos alunos uma vivência de etapas inerentes ao processo de desenvolvimento de um jogo educativo que agregou para além do entretenimento, a aquisição de conhecimentos necessários na edificação de jovens comprometidos com a sociedade por meio da tecnologia. O jogo foi desenvolvido no motor de jogos *Unity 3D* e será disponibilizado para sistemas *Android*, evidenciando os processos de tomadas de decisão e otimização no decorrer da implementação do projeto.

Palavras-chave: jogos sérios, *design* de jogo, desenvolvimento de jogo, Unity, controle social.

ABSTRACT

The focus of the work presented here considers the growing gaming market, which currently establishes a relation with the world of entertainment, concomitant with the universe of serious games, an increasingly significant reality in the evolution of a society more conscious and able to discern values and social control, a context that consolidates every day. For that, a vision was presented that contemplated the work of design and development of the game "Recruta social", a project idealized by a group of students of the Federal University of Paraíba in partnership with the TCE-PB and gave the students an experience of stages inherent in the process of developing an educational game that added beyond the entertainment, the acquisition of knowledge necessary in the edification of young people committed to society through technology. The game was developed in the Unity 3D game engine and will be made available to Android systems, highlighting the decision-making and optimization processes in the course of project implementation.

Key-words: serious games, game design, game development, Unity, social control.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE CÓDIGO

LISTA DE ABREVIATURAS

API	–	*Aplication Program Interface*
CPU	–	*Central Processing Unit*
ECS	–	*Entity Component System*
ESA	–	*Entertainment Sofwtare Association*
FPS	–	*Frames per Second*
GC	–	*Garbage Collection*
GDD	–	*Game Design Document*
GPU	–	*Graphic Processing Unit*
GUI	–	*Graphic User Interface*
IHC	–	Interação Homem-Computador
IJD	–	Indústria de jogos digitais
ISO	–	*International Organization for Standardization*
LAViD	–	Laboratório de Aplicações de Vídeo Digital
NPC	–	*Non-Playar Character*
PGB	–	Pesquisa Game Brasil
PWC	–	*PriceWaterhouseCooper*
TCE	–	Tribunal de Contas do Estado
TCE-PB	–	Tribunal de Contas do Estado da Paraíba
UI	–	*User Interface*

SUMÁRIO

1 INTRODUÇÃO

A indústria de jogos digitais promove diversas inovações tecnológicas para diversos setores, como: arquitetura e construção civil, marketing e publicidade, áreas de saúde, educação, treinamento e capacitação.

Atualmente, o público dos usuários de jogos digitais não consiste apenas em jovens do sexo masculino, como se pensa tradicionalmente, mas também de mulheres, crianças e idosos. Segundo a Pesquisa Game Brasil (PGB) 2018, as mulheres correspondem a 58,9% do público brasileiro que joga. Por outro lado, o mercado de trabalho brasileiro na área de *games* ainda é predominantemente masculino, embora os dados levantados pelo 2° Censo da Indústria Brasileira de Jogos Digitais (ROSA, 2018) demonstrem que o número de mulheres triplicou no período entre 2013 a 2018, correspondendo a 20,7% dos profissionais.

De acordo com o levantamento realizado pela consultoria *PricewaterhouseCooper* (PCW), em 2014, o mercado de jogos digitais movimentou cerca de US$ 65,7 bilhões em 2013, e projeta uma taxa de crescimento anual de 6,3%. Em comparação com o mercado de filmes, que movimentou US$ 88,2 bilhões no mesmo ano, possui uma projeção de taxa de crescimento anual de 4,5%. Estimando que, em 2018, a indústria de jogos ultrapasse a indústria de filmes de entretenimento (FLEURY; NAKANO; CORDEIRO, 2014, p.33).

No Brasil, estimava-se que o mercado de *games* teria um crescimento de US$ 480 milhões, em 2013, para US$ 840 milhões, em 2018, correspondendo a uma taxa de crescimento anual de 13,5% (FLEURY; NAKANO; CORDEIRO, 2014). Porém, de acordo com os dados da Newzoo (PACHECO, 2018), o mercado de *games* no Brasil movimentou cerca de US$ 1,3 bilhão em 2017, envolvendo em torno de 66,3 milhões de jogadores, colocando o Brasil em 13° no *ranking* mundial e em primeiro lugar na América Latina. Além disso, mesmo com o país em crise econômica, foi previsto o crescimento do número de jogadores no Brasil para 75,7 milhões, em 2018, podendo movimentar um valor em torno de US$ 1,5 bilhão (Dino, 2018).

Para Fleury, Nakano e Cordeiro (2014), o crescimento do mercado de jogos digitais ocorre devido aos avanços tecnológicos, com o aumento do poder de

processamento do *hardware*, capacidade gráfica das *GPU*s, mais expansão e barateamento da internet e banda larga. Além disso, os *smartphones* exercem um grande papel nesse crescimento, pois no Brasil há, em média, mais de um *smartphone* ativo por habitante (cerca de 1,5), de acordo com a pesquisa da GVcia (Meirelles, 2018), sendo o principal meio de acesso à internet de 92% da população brasileira, segundo o IBGE (Paypal, 2018). Como resultado, a pesquisa levantada pelo SuperData e Paypal (2018) destaca que 82% dos brasileiros jogam através de seus *smartphones*, isso pode ocorrer devido aos baixos custos destes jogos e a facilidade de acesso proporcionada pelos *smartphones*. Por fim, de acordo com a Newzoo, os *smartphones* são responsáveis por 51% (cerca de US$ 70,3 bilhão) da receita global da indústria dos *games* (Paypal, 2018).

Dito isso, os resultados apontados pelo 2° Censo da Indústria Brasileira de Jogos Digitais (ROSA, 2018) apresentam um crescimento de empresas no Brasil, que produzem *games*, passando de 142 para 375 no período entre 2013 a 2018. Segundo a pesquisa do *Homo Ludens* (ROSA, 2018), foram produzidos mais de 1700 jogos, em um período de 2 anos, sendo 43% para dispositivos móveis, 24% para computadores, 10% para plataformas de realidade virtual e aumentada e 5% para os consoles. Dentre esses jogos, cerca de 51% foram classificados como educativos e o restante para entretenimento.

Logo, é possível afirmar que os jogos digitais contemplam vários setores, além do entretenimento, podendo adquirir um caráter "sério", incorporando atividades de educação, pesquisa científica ou capacitação de profissionais em diversas áreas, tais como: saúde, arquitetura e construção civil, esses jogos são comumente denominados de *serious games*. De acordo com Ritterfeld, Cody e Vorderer (2009), a quantidade de profissionais interessados em usar jogos para educar, motivar e incentivar os jogadores, cresceu em um curto período. Este movimento teve grande crescimento com o surgimento de portais e *websites*, como o "*Games for Change*[1]" e o "*Games for Health*[2]", que apresentam e incentivam este tipo de ferramentas educacionais, abrindo espaço para várias discussões e chamando a atenção de *game designers*, educadores e acadêmicos.

Algumas propriedades de jogos, como interatividade e apresentação de regras e desafios, podem proporcionar experiências e diversão que motivam o usuário a

[1] Disponível em: http://www.gamesforchange.org/
[2] Disponível em: https://www.gamesforhealtheurope.org/

continuar jogando. Desta forma, os *serious games* podem ser um meio bastante eficiente para educação, desenvolvendo competências e atitudes, as quais podem ser desempenhadas em alguma situação real, bem como a construção do conhecimento sobre variados temas, relacionados à saúde, arquitetura e até políticas sociais.

1.1 Problema

O controle social pode ser entendido como a participação do cidadão na gestão pública realizando fiscalização, monitoramento e controle das ações da administração pública. Ele é exercido pela própria sociedade e tem como principais objetivos a prevenção e combate à corrupção, a aproximação com o poder público e o exercício da cidadania. Contudo, devido à falta de cultura participativa e fiscalizatória, o controle social não é ainda praticado de forma efetiva pela sociedade.

Isso se deve, muitas vezes, pelo desconhecimento de diversos conceitos relacionados com a própria administração. De fato, são imensos os desafios em motivar e criar uma cultura capaz de conduzir a população a identificar possibilidades de mudanças, na condução da administração pública, através de sua participação no controle social.

Sendo assim, o problema abordado nesta proposta é fazer a instrução deste conteúdo sobre controle social e órgãos de controle, especificamente o Tribunal de Contas do Estado, tendo como instrumento ou meio um jogo digital educativo e como público alvo crianças e adolescentes na faixa de 10 a 15 anos.

1.1.1 Objetivo geral

O objetivo deste trabalho consiste em apresentar a concepção e o desenvolvimento do jogo “Recruta Social”, que promove o papel do TCE e o controle social, focando principalmente em aspectos de *game design*, boas práticas de implementação e otimização do projeto.

1.1.2 Objetivos específicos

No intuito de alcançar o objetivo geral supracitado, os seguintes objetivos específicos foram abordados neste trabalho:

- Compreender o conceito de controle social e o papel do TCE;
- Mapear o conteúdo instrucional em uma proposta de jogo;
- Apresentar o processo de concepção sobre aspectos do *game design*, como mecânicas e narrativas;
- Expor o processo de desenvolvimento do jogo;
- Evidenciar as estratégias e o processo de otimização do "Recruta Social";

1.2 Estrutura da monografia ou TG

Este trabalho foi dividido em 9 capítulos, os quais estão organizados da seguinte maneira:

- O Capítulo 1 corresponde à introdução deste trabalho;
- O Capítulo 2 representa a fundamentação teórica, apresentando alguns conceitos chaves para a compreensão do trabalho, como *game design*, *serious games*, conceitos de computação gráfica, assim como fundamentos de desenvolvimento na *Unity*;
- O Capítulo 3 apresenta a proposta do jogo, apresentando uma breve descrição sobre suas propriedades;
- O Capítulo 4 explica o processo de concepção e tomadas de decisões para o desenvolvimento do jogo;
- O Capítulo 5 demonstra os princípios e boas práticas aplicadas na codificação do jogo em questão;

- O Capítulo 6 destaca as estratégias aplicadas para a otimização da aplicação;
- O Capítulo 7 aponta alguns desafios encontrados no processo de desenvolvimento;
- O capítulo 8 apresenta os resultados obtidos na otimização do jogo;
- O capítulo 9 encerra o trabalho apresentando uma conclusão e as possibilidades para um trabalho futuro.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Projetando e desenvolvendo jogos digitais

Nesta seção será discutido alguns conceitos e principais características dos jogos eletrônicos e o seu desenvolvimento, falando sobre o papel do *game designer*, os elementos básicos de um *video game*, além da necessidade de documentação e implementação de protótipos.

2.1.1 Sobre os jogos digitais

Antes de falar sobre o processo de desenvolvimento de *games*, é importante ter uma compreensão clara e bem definida sobre o que eles são e as suas principais características.

Para Juul:

> Um jogo é um sistema formal baseado em regras; cujos resultados são variáveis e quantificáveis; onde diferentes variáveis determinam diferentes valores; o jogador exerce o esforço a fim de influenciar o resultado, o jogador sente-se unido ao resultado; e as consequências da atividade são opcionais e negociáveis. (2005 apud SATO, 2010, p. 75).

Segundo a definição feita por Schell (2008), "Um jogo é uma atividade de solução de problemas, conduzindo atitudes lúdicas" (tradução nossa). Desta forma, os jogos são capazes de proporcionar experiências prazerosas, através da solução do problema em um tema específico, como espaciais, medievais, pirataria, entre outros. Com isso, podemos reconhecer algumas qualidades atribuídas aos jogos digitais. O autor ainda afirma que os problemas a serem resolvidos podem ser vistos como os objetivos do jogo, os quais podem ser a fonte de motivação do jogador; o conjunto de métodos permitidos para solucionar o problema consiste nas regras do jogo; os caminhos para solucionar o impasse podem apresentar obstáculos e envolver vários conflitos;

Além disso, Schell (2008) apresentou uma lista de qualidades presentes nos jogos digitais, que serão apresentadas no Quadro 1.

Quadro 1 – Qualidades presentes nos jogos digitais

Qualidade	Descrição
Jogos são sistemas formais e fechado	Os jogos são representações limitadas da realidade e são compostos por partes inter-relacionadas que funcionam em conjunto, correspondendo a um sistema. Além disso, são formados por regras bem definidas (formal) e autocontidas dentro do jogo (fechado).
Jogos são atividades voluntárias	O jogador pode usufruir do jogo por sua própria vontade e por quanto tempo quiser.
Jogos possuem objetivos	Correspondem aos problemas nos quais o jogador deve resolver.
Jogos possuem conflitos	Ocorrem através da interação do jogador em busca de solucionar o problema.
Jogos possuem regras	Consistem em restrições aos meios de alcançar os objetivos, definindo a forma como o jogador pode agir.
Jogos podem ser vencidos ou perdidos	O jogador pode vencer ou perder para os desafios proporcionados pelo jogo.
Jogos são interativos	Indica que o jogador é um agente ativo, podendo gerar causas e efeitos na representação de mundo do jogo.
Jogos possuem desafios	Corresponde ao conjunto de obstáculos que o jogador deve superar para alcançar os objetivos do jogo. Bons jogos

	apresentam desafios balanceados, não sendo muito fácil e nem muito difícil.
Jogos possuem valores internos	Implica em elementos valiosos que possuem algum valor apenas dentro do jogo, como moedas e experiência.
Jogos estimulam os jogadores	Consiste na capacidade dos jogos imergirem jogadores em seu universo.

Fonte: Schell (2008)

2.1.2 Elementos básicos de um jogo

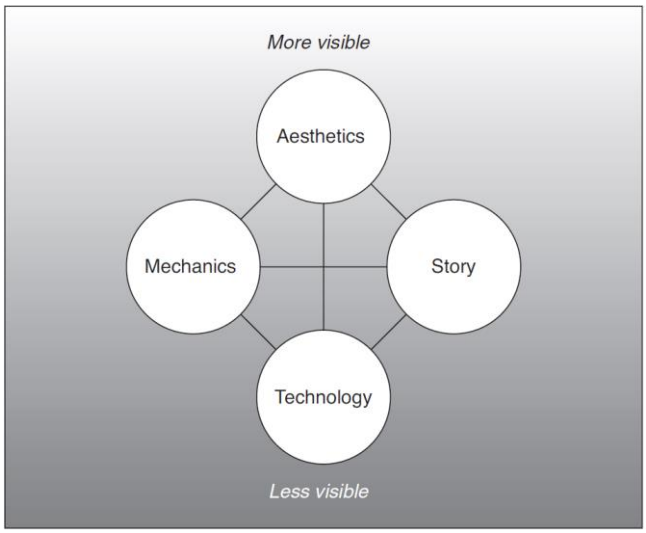
Um dos principais aspectos necessários para desenvolver jogos é compreender como eles funcionam, quais são os seus principais elementos e como estes são capazes de despertar o interesse do jogador e fornecer uma experiência memorável.

Para Schell (2008), os *games* são compostos por 4 elementos fundamentais, descritos em um modelo, denominado por ele como "tétrade elementar" (Figura 1**)**. Sendo eles:

- Mecânica: consiste nas regras e conceitos que definem os processos e os objetivos do jogo;
- Estética: representa os aspectos audiovisuais de um jogo, apresentando uma relação muito próxima com a experiência e estímulos do jogador;
- História: consiste na sequência de eventos os quais ocorrem no universo do jogo;
- Tecnologia: corresponde aos meios que possibilitam a execução e interação do jogo;

Este modelo está organizado em formato de diamante, indicando que nenhum elemento é mais importante do que o outro, ele apresenta os componentes em um "gradiente de visibilidade", no qual o elemento de estética é o mais evidente para o jogador, o elemento de tecnologia é o menos visível, e os componentes de história e mecânica estão em níveis intermediários.

Figura 1 – Tétrade elementar



Fonte: Schell (2008)

2.1.2 Gêneros de jogos

O mercado de *games* é composto por uma grande diversidade de aplicações, com características distintas, com objetivos e mecânicas específicos, havendo a necessidade de classificá-los, a partir de suas características, em gêneros estabelecidos pelo mercado. É possível também que um jogo possa pertencer a mais de um gênero ou combinação deles. Vince (2018) apresenta uma descrição para diversas categorias de jogos digitais, como pode ser visto no Quadro 2.

Quadro 2 – Gêneros de jogos digitais

Categoria	Descrição
Ação	Jogos de ação colocam o jogador no controle e centro da ação, apresentando desafios físicos que ele deve superar.
Ação-aventura	Jogos de ação-aventura incorporam dois tipos de mecânicas: missões e obstáculos que devem ser conquistados com a coleta/uso de um determinado item; além

	de elementos de ação, os quais utilizam determinados itens.
Aventura	Em jogos de aventura, o jogador interage com o ambiente e personagens, solucionando *puzzles* para a progressão da história ou *gameplay*.
RPG	*Role-Playing Games* apresenta como principais mecânicas: realização de missões (*Quests*), exploração de cenário, customização de personagem, aprimorar os atributos do avatar do jogador, além de permitir tomadas de decisões que alteram o rumo da história.
Simulação	Consiste em programas que buscam emular a realidade, simulando situações e eventos reais.
Estratégia	Jogos de estratégia dão ao jogador acesso do mundo e seus recursos, além de exigir uma elaboração cuidadosa de estratégias e táticas para superar os desafios.
Esportes	Simulam esportes reais como futebol, basquete, *golf*, corrida, entre outros.
Puzzle	Jogos em que o jogador deve solucionar problemas para progredir no *game*.
Casual	Jogos casuais apresentam mecânicas básicas e repetitivas, que são ideias para pequenas sessões de jogatina.

Educativo	Ferramentas de aprendizagem utilizadas para ensinar um determinado assunto através de mecânicas de jogo.

Fonte: Vince (2018)

Dentre esses gêneros apresentados, destaca-se neste trabalho os jogos educativos, que serão apresentados em mais detalhes no seguinte tópico.

2.1.2.1 Jogos educativos

Muitas mídias podem ser consideradas ferramentas para os propósitos educacional e de entretenimento, a exemplo de vídeos, programas de televisão, música, quadrinhos e, mais recentemente, os *serious games*. Podendo atrair os usuários ao ensinar habilidades e determinados assuntos, especialmente se apresentar aspectos que mantenham o usuário usufruindo do conteúdo, com uma boa narrativa ou um mundo vívido.

O *Serious games* são jogos com propósitos educacionais para qualquer faixa etária e situações diversificadas. Em 1987, Clark Abt estabeleceu esta terminologia, indicando que os *serious games*:

> Facilitam a comunicação de conceitos/fatos – devido a dramatização de problemas e motivação – além de contribuírem para o desenvolvimento de estratégias, a tomada de decisão, o desempenho de papeis, dentre outras vantagens, em um ambiente em que o feedback é instituído de maneira ágil (ABT, 1987 apud FLEURY; NAKANO; CORDEIRO, 2014, p. 70).

O termo em questão pode ser mal interpretado pelo seu significado literal, o qual seria um oximoro: segundo Newman (Ritterfeld; Cody; Vorderer, 2009, p. 3), jogos são intrinsicamente divertidos e não sérios. Apesar desta contradição aparente, estudiosos e profissionais enxergam que *serious games* podem ser divertidos, ao mesmo tempo que são educacionais, tendo um propósito impactante e significativo. Klopfler et al. (2010 apud FLEURY; NAKANO; CORDEIRO, 2014, p. 70) afirma que quaisquer jogos digitais utilizados com proposta de aprendizagem, independente de terem sido criados com essa intenção, são *serious games*.

De acordo com Ritterfeld, Cody e Vorderer (2009), os jogos sérios são desenvolvidos, a partir dos seus valores de entretenimento, além de adicionar um

componente educacional, com isso eles se tornam um gênero designado a ter um propósito extra, agregando conhecimento, para além da diversão. A partir dos argumentos supracitados, infere-se que os jogos sérios buscam a aprendizagem por meio das experiências satisfatórias e podem abordar diversas áreas. Desta forma, muitos elementos de jogos de entretenimento somam-se aos *serious games*, não obstante a ideia de que o ato de jogar possui uma motivação própria, por fornecer também entretenimento.

O resultado educacional bem-sucedido de um *serious game*, cuja as informações estão embutidas em sua programação de entretenimento, depende de alguns fatores relevantes, apontados por Ritterfeld, Cold e Vorderer (2009), tais como: capturar a atenção dos usuários, garantindo que a temática e as ações a serem adotadas sejam discutidas repetidamente ou exibidas visualmente, demonstrado as suas vantagens e recompensas; é importante que uma história seja envolvente, possuindo personagens carismáticos ou outros elementos que possam motivar o usuário a procurar mais informações. Se um desses elementos estiver faltando, o resultado educacional pode ser comprometido.

Com o propósito de unir o aspecto educacional com a experiência de entretenimento, Ritterfeld, Cold e Vorderer (2009) afirmam que é necessária uma experiência paralela, isto é, o componente de educação deve ser divertido ao mesmo tempo que o componente de entretenimento deve estar fortemente associado à educação.

2.1.3 Game Design

Segundo Schell (2008), o *Game Design* consiste em um processo de tomada de decisões a respeito de como o jogo deve ser e a experiência (*gameplay*) que este deve prover ao jogador. Esse processo é realizado de forma iterativa, ou seja, esta tarefa ocorre várias vezes durante o processo de desenvolvimento, podendo ser tomadas centenas ou milhares de decisões, desde a parte inicial do projeto, até as etapas mais avançadas do desenvolvimento.

Para Brathwaite (2009 apud SATO, 2010, p.75), *Game Design* é o processo de criar disputas e as regras de um jogo. Sendo necessário elaborar objetivos que motive o

jogador a alcançá-los e regras que precisam ser seguidas por ele, ao fazer escolhas as quais alteram o resultado do jogo em prol do objetivo.

Para Sato:

> O Game Design deve trazer em sua essência, as características necessárias para o jogador sentir-se unido ao contexto apresentado pelo jogo, podendo assim realizar escolhas e tomar decisões pertinentes para o progresso no jogo. Isto é, o game design deve oferecer oportunidades para os jogadores, que os levem a realizar decisões que afetarão o resultado do jogo. (SATO, 2010, p.76).

Segundo Schell (2008), o principal foco do *game designer* consiste em criar experiência para o jogador, por meio das mecânicas (regras e objetivos), da dinâmica (execução das mecânicas) e estética do jogo. Sendo um dos principais motivos que faz alguém jogar e interagir com determinado *game*.

Deste modo, o *game designer* toma decisões com foco no jogador, desde o início do processo criativo, definindo as características básicas, como sua estrutura e funcionamento, grau de dificuldade e progressão do jogo. É importante ressaltar que o *designer* trata de um papel e não estritamente de uma pessoa, desta forma, um criador de conteúdo, assim como artistas ou narradores, que realiza decisões criativas de como o jogo deve ser, também são *game designers*.

2.1.4 Habilidades de *Game Designer*

Embora possa se pensar que um *game designer* precise ter um vasto conhecimento sobre tecnologia, artes e narrativas para realizar as suas decisões, ainda é possível executá-las sem saber todos os detalhes técnicos pertinentes ao processo. Mas isso não descarta a importância de possuir conhecimento e habilidades em diversas áreas do desenvolvimento de um *game*, uma vez que elas podem permitir o *designer* a ter ideias melhores e tomar decisões mais conscientes.

De acordo com Schell (2008), alguns conhecimentos e habilidades são importantes para um *game designer*, tais como:

- Animação: Permite trazer movimentos e vida ao ambiente do jogo;

- Antropologia: Para entender os desejos do público alvo;
- Arquitetura: Permite construir cenários e mundos complexos e entender a relação entre pessoas e espaço (*Level Design*);
- Arte visual: Importante para a visualização e construção dos elementos gráficos do jogo;
- *Brainstorm*: Para criação de novas ideias;
- Comunicação: Para poder conversar com pessoas de várias áreas de desenvolvimento, solucionar conflitos e conhecer melhor a equipe de desenvolvedores, o cliente e os jogadores;
- Engenharia: Fornece uma visão mais aprofundada dos limites e potência que uma tecnologia pode trazer, além de permitir se adequar as inovações tecnológicas;
- Gerência: Necessário para todo trabalho em equipe em busca de um objetivo;
- Matemática: Jogos são compostos por matemática, tanto na área da computação gráfica quanto na área da ciência da computação;
- Música: Fundamental para comover e imergir o jogador;
- Psicologia: Entender o psicológico humano ajuda a criar experiências mais envolventes;

Mesmo que essas competências sejam de grande importância para a tomada de decisão, elas podem ser encontradas no conjunto de habilidades da equipe. Desta forma, para Schell (2014), a habilidade mais importante é ser um bom ouvinte, ou seja, é fundamental o *designer* estar sempre disposto a escutar alguém antes de tomar alguma decisão, uma vez que o coletivo costuma fazer parte do desenvolvimento de um jogo: o *designer* está produzindo um jogo para uma audiência de várias pessoas; além de estar trabalhando em conjunto com uma equipe, a fim de possibilitar uma diversidade maior de ideias.

Além disso, segundo Schell (2008), é preciso escutar os consumidores para satisfazê-los, pois eles serão as pessoas as quais irão jogar e ter experiências com o *game*, portanto é fundamental a compreensão dos seus pensamentos, desejos e emoções para o desenvolvimento de um jogo bem-sucedido. É necessário também, escutar os clientes,

uma vez que eles estão pagando pelo produto que deve atender as suas exigências. E, por fim, é importante que o *designer* escute a si mesmo, através da introspecção, procurando dar espaço para a sua criatividade e conhecimentos.

2.1.5 Documentações de jogo

A documentação de um jogo agrupa várias especificações do projeto, como elementos da mecânica (objetivos e regras), gráficos, narrativa, interface. É um artefato bastante importante para manter a comunicação e memória das decisões tomadas durante o processo de desenvolvimento do jogo, podendo evitar conflitos e perdas de ideias elaboradas e trabalhos realizados.

Para Schell (2008), a grande necessidade dos documentos de um jogo está nas limitações da memória humana, que nem sempre é capaz de manter as decisões e boas ideias por muito tempo, e a urgência de manter a equipe do projeto informada sobre as decisões de *design* e especificações do jogo, permitindo que a equipe esteja mais apta a participar do processo de *design* e encontrando falhas com maior facilidade.

Vários tipos de documentos podem ser elaborados durante o desenvolvimento de um jogo, conhecidos por: documento de *design*; documento de engenharia; documento de gerência de projeto; e documento de narrativa. Dentre eles, destaca-se o documento de *design*, tendo em vista a sua relevância considerando a temática do trabalho aqui exposto.

2.1.5.1 Documento de *design* do jogo

O documento de *design* do jogo (*Game Design Document* – GDD) descreve uma visão geral sobre o *design* do *game*, permitindo uma compreensão sobre os objetivos do jogo, sem entrar em muitos detalhes. Não obstante, este documento pode registrar detalhes sobre as mecânicas e interfaces do *game*, permitindo que os *designers* possam memorizar os detalhes de suas ideias e decisões, além de comunicar estes detalhes para as equipes que serão responsáveis por implementá-los. E, por fim, podem abordar também uma breve descrição da narrativa do jogo, apresentando os personagens, ações e eventos os quais vão ocorrer dentro da história.

2.1.6 Prototipagem

De acordo com Paavilainen (apud SATO, 2010, p.77), um protótipo no desenvolvimento de jogos consiste em uma amostra do resultado final, desenvolvida em um curto período, apresentando algumas características e funcionalidades do jogo. Este protótipo pode ser crucial para demonstrar, avaliar e testar aspectos cruciais do produto, antes de finalizá-lo.

Para Sato (2010), a importância de protótipos para o processo de criação, desenvolvimento e produção de jogos é descrita pelos seguintes motivos:

- Permite uma interação prévia das ideias e decisões do *game design*;
- Oportuniza a identificação de novas possibilidades para o produto;
- Viabiliza os teste e validações de conceitos e ideias;
- Possibilita testar as ações e limites espaciais do jogo;
- Oportuniza o estudo de balanceamento das mecânicas;
- Oferece redução de custo e risco;
- Permite o *feedback* do público alvo, antes do lançamento do jogo.

Esse mecanismo de desenvolvimento de protótipos é bastante viável no desenvolvimento de jogos, uma vez que corresponde a um processo iterativo, isto é, ocorre de forma cíclica, em que se define, implementa, testa e valida várias características e funcionalidades do jogo de forma contínua. Permitindo desenvolver o jogo de modo progressivo e controlado, sem precisar implementar todos os elementos de uma vez.

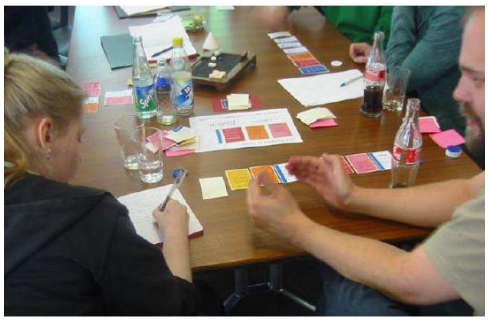
De acordo com Sato (2010), a construção de protótipos pode ocorrer em várias etapas do desenvolvimento do jogo, desde concepção até a implementação do produto, podendo ser construídos modelos funcionais físicos ou digitais. Esses modelos consistem em protótipos que apresentam o funcionamento geral das mecânicas do projeto e devem ser jogáveis.

Desta forma, é possível construir protótipos físicos (Figura 2), com cartolinas, papeis, lápis, entre outros materiais, podendo ser mais fácil e rápido de se construir, uma vez que pode ser mais informal e não exige contato com a elaboração de código. Para

Fullerton (2008 apud SATO, 2010, p. 79), este tipo de protótipo permite focar mais no funcionamento da mecânica do que na tecnologia na qual o jogo funcionará.

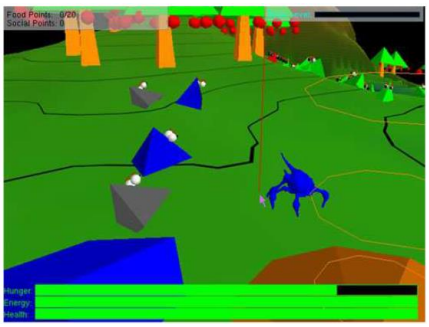
Já os protótipos digitais (Figura 3) são representações simplificadas em *softwares* do produto a ser desenvolvido na sua tecnologia alvo, podendo proporcionar uma ideia do funcionamento geral das mecânicas; realizar testes de situações específicas do jogo; avaliar as limitações e viabilidade da implementação do jogo na tecnologia adotada.

Figura 2 – Protótipo físico



Fonte: Sato (2010)

Figura 3 – Protótipo digital



Fonte: Sato (2010)

2.2 Conceitos de computação gráfica

Os dados levantados pela ESA (2018), apontam que os elementos visuais (gráficos) são um dos principais fatores os quais influenciam a decisão de compra de jogos eletrônicos. Com isso, é importante ter uma base de conhecimento sobre computação gráfica, uma vez que ela é a fundamentação para a criação de gráficos realistas (ou estilizados) e ambientes virtuais.

Neste tópico, será abordado alguns conceitos e técnicas aplicadas na computação gráfica de jogos digitais, que facilitam o entendimento de tópicos os quais serão discutidos mais adiante, neste trabalho.

2.2.1 O que é computação gráfica?

Antes de apresentar qualquer conceito ou técnica de computação gráfica, devemos primeiramente entender o que ela é e o porquê desta área da computação ser tão almejada em várias indústrias, como automobilismo, entretenimento, arquitetura, medicina, entre outras.

Segundo a ISO (*International Organization for Standardization*), computação gráfica é: “Um conjunto de ferramentas e técnicas para converter dados para ou a partir de um dispositivo gráfico através do computador”. Em outras palavras, seria um conjunto de algoritmos e equações que converteria determinados dados em uma representação visual, seja ela artística ou científica.

Nas palavras de Hughes et al. (2014), a computação gráfica é um campo multidisciplinar no qual a física, matemática, interação homem-computador (IHC), engenharia, arte e design gráfico desempenham um papel importante. Com a física, o computador é capaz de simular a luz, a interação entre objetos dentro do espaço tridimensional e as animações entre personagens. Com a matemática, é possível descrever formas de objetos, realizar operações bidimensionais e tridimensionais como translação e rotação. Engenharia é capaz de expandir o uso de memória e reduzir o tempo de processamento, otimizando os processos de computação gráfica. E por fim, a arte é impulsionada, através da IHC, permitindo uma comunicação de computador para humano mais eficiente e compreensiva.

Segundo Azevedo (2003), a computação gráfica para a síntese de imagens consiste na representação visual de objetos criados por computadores, a partir das representações geométricas e visuais de seus componentes. Essa área pode ser compreendida também como visualização computacional ou científica, em que há uma preocupação de gerar uma informação visual, para facilitar o entendimento de conjunto de dados complexos, como, por exemplo, dinâmica de fluídos ou simulação espacial.

Muitas vezes, a computação gráfica está intimamente relacionada com a arte e pode ser vista como uma ferramenta que possibilita artistas a aplicarem técnicas de desenho e modelagem, através do computador. Imagens que antes exigiriam muita

habilidade e técnicas de um pintor, agora podem ser simplificadas e auxiliadas pela máquina, como por exemplo, a representação da luz e sombra em uma cena.

Considerando a grande difusão dos computadores na vida moderna e o fato de que a visão é um dos sentidos mais utilizados para a nossa percepção de mundo, a existência da computação gráfica garante uma ampla gama de aplicações nas mais diversas áreas. Uma vez que ela vai além de ser um conjunto de ferramentas e equações que produzem efeitos visuais, ela traduz uma nova forma de produzir arte e representações do mundo a nossa volta. Além de capacitar e auxiliar várias pessoas nas produções visuais, o que antes exigiria muita habilidade artística para ser produzida à mão. Por conseguinte, os avanços tecnológicos, com processadores e placas gráficas mais sofisticadas permitiram que a computação gráfica evoluísse cada vez mais, expandindo a sua utilização em diversas áreas.

2.2.2 Sistema visual humano

A visão humana é o sentido mais essencial para a computação gráfica, uma vez que não seriamos capazes de detectar a mensagem transmitida pela máquina, através dos gráficos gerados por ela sem este sentido. Sendo assim, entender como interpretamos as imagens bidimensionais e percebemos profundidade é de grande importância para a confecção de imagens digitais.

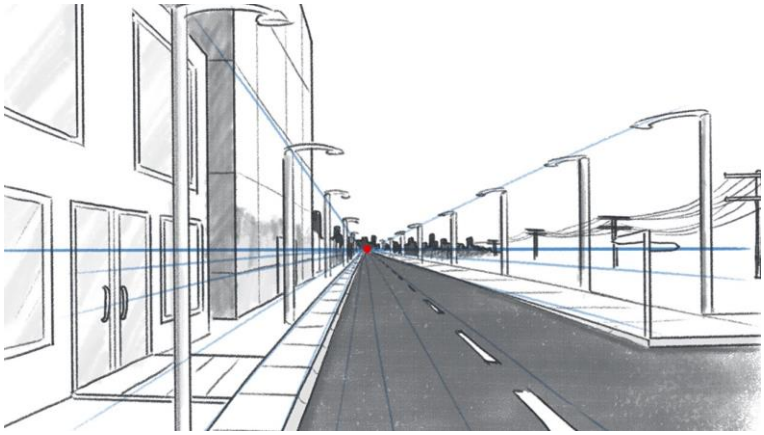
Nas palavras de Hughes et al. (2014), o sistema visual humano é capaz de realizar uma série de tarefas de forma bastante eficiente, como reconhecer formas, tamanhos, orientação e cores em diferentes níveis de luz. Em contrapartida, acaba realizando outras tarefas de forma imprecisa quando envolve alterações sutis, como detectar se duas linhas com angulações próximas são paralelas, identificar brilho absoluto ou identificar se duas cores não adjacentes são iguais.

A percepção espacial é a capacidade da visão humana de detectar e distinguir objetos de acordo com seus formatos, cores, texturas e as relações espaciais entre eles. Algumas informações surgem, a partir da percepção espacial, como perspectiva e oclusão.

A perspectiva é uma percepção que obtemos com a relação entre tamanho dos objetos observados e a distância relativa entre eles e o observador, que causa o efeito de redução de tamanho do objeto, à medida que este vai se distanciando do observador. Em outras palavras, percebemos objetos mais próximos em uma proporção maior e objetos distantes em uma proporção menor.

O conhecimento prévio do objeto, alinhado à perspectiva, nos permite identificar tanto a distância de um objeto, a partir do observador, como as distâncias relativas entre os objetos dentro do campo de visão, uma vez que conheceríamos o tamanho destes objetos, possibilitando determinar quais deles estão mais distantes ou mais próximos. Esta informação é amplamente utilizada em artes e desenhos para adicionar maior realismo e semelhança à visão humana, como pode ser visto na figura 4.

Figura 4 – Visão em perspectiva



Fonte: Canal do *Youtube* "*Draw with Jazza*"[3]

Além da perspectiva, a oclusão carrega informações a respeito das posições relativas das coisas, já que ela ocorre quando um objeto opaco ou translúcido obstrui a visão de outro objeto, que está mais afastado do observador. Desta forma, se um determinado objeto obscurece um outro, seremos capazes de identificar que este objeto está mais próximo do observador em relação ao outro.

Na figura 5, pode-se perceber algumas informações de localidade a partir da oclusão, como por exemplo: o cachorro se encontra a frente da parede; a cadeira está atrás

[3] Disponível em: https://www.youtube.com/watch?v=b4Le5qsIBqQ

da porta, pois ela está sendo obstruída pela parede; e a vassoura está parcialmente na frente da caixa no chão, o que indica que ela está mais próxima do observador.

Figura 5 – Demonstração de oclusão

Fonte: Fala Atelier

2.2.3 Rendering

Como discutido anteriormente, a computação gráfica é utilizada para síntese de imagem com suporte em um computador. Esse processo de criação de uma imagem computacional é denominado *rendering* e ocorre através dos dados e informações dos objetos os quais compõem esta imagem, como a sua geometria, material, texturas, posição espacial e sombreamento.

O processo de *rendering* pode ser interpretado como a conversão de dados, que descrevem uma cena virtual, em uma imagem realista de modelos 2D ou 3D. O nível de realidade deve ser adequado ao seu tipo de uso, tendo em vista que aumentar o realismo de uma cena ocasionará um maior tempo de processamento e custo de geração.

Segundo Azevedo (2003), o processo de geração de imagens realistas é incremental, pois é realizado de forma repetida, combinando resultados passados com os resultados atuais. Além disso, o processo é composto por 7 fases que serão descritas a seguir:

- Construção de modelo consiste na utilização de técnicas de modelagem para a construção de cenas virtuais;

- Aplicação de transformações lineares (como projeções) aos objetos, fazendo com que possuam uma aparência tridimensional em dispositivos de visualização bidimensionais, como a tela do computador;
- Eliminação de polígonos ou faces escondidas (*culling back-faces*) devido à sua posição relativa entre os objetos da cena e o observador. Esta técnica pode ser refeita à medida que o ponto de observação ou objetos da cena são reposicionados;
- Utilização da técnica de *clipping*, consiste em desconsiderar partes da cena que não serão mostradas, como por exemplo, objetos obstruídos por outros, detalhes de objetos muito distantes do observador ou muito pequenos;
- Rasterização da cena, consiste em um processo de conversão dos dados tridimensionais dos modelos em pixels, isto é, converter linhas e áreas contínuas em coordenadas de pixels, para serem exibidas em uma tela;
- Eliminação de partes de objetos de acordo com sua posição relativa aos demais, removendo partes que estejam sendo obstruídas do campo de visão. Essa fase é uma continuação da 3ª, porém ocorre depois do processo de rasterização;
- Coloração dos pixels, realizada de forma incremental ou interpolada. Nessa etapa, o realismo "fotográfico" da cena começa realmente a ser tratado e percebido, podendo ser necessário o uso de modelos físicos para representar a luz, sua direção e intensidade na cena, projeções de sombras e propriedades de materiais, como reflexão, transparência, brilho.

2.2.3.1 Taxa de quadros

Em aplicações 3D de tempo real como jogos eletrônicos, várias imagens virtuais (*frames*), são geradas uma certa quantidade de vezes por segundo. Essa métrica é denominada de taxa de quadros (*frame rate*), que indica o número de *frames* os quais são gerados por segundo, sendo um indicador de fluidez na sequência de imagens utilizadas para dar a sensação de movimento.

2.2.4 Transformações geométricas

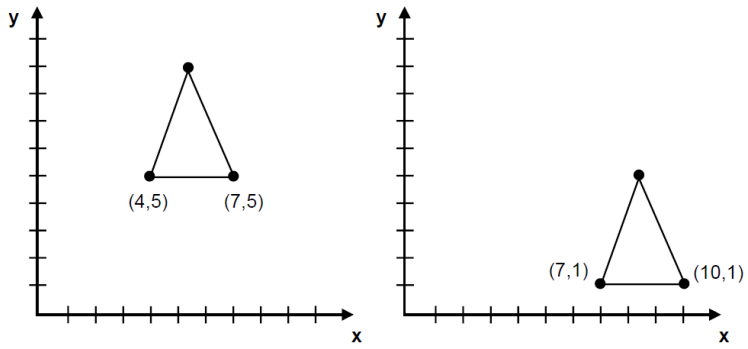
A utilização de operações de transformações em pontos e objeto são de grande relevância para jogos eletrônicos, pois permitem que elementos na cena possam ter um determinado comportamento espacial. Como por exemplo, fazer um personagem se mover no cenário.

Realizar transformações em objetos, nada mais é do que realizar operações matemáticas em cada um dos pontos que compõem esse objeto. Neste tópico, serão descritas algumas das operações geométricas básicas, como translação, escala e rotação.

2.2.4.1 Transformação de translação

A translação consiste no deslocamento dos pontos de um objeto, ou forma geométrica, de uma coordenada (*x*,*y*) para outra coordenada (*x'*, *y'*). De forma resumida, esta operação pode ser interpretada como mover um objeto de uma posição A para uma posição B, como mostrada na figura 6.

Figura 6 – Translação de objeto



Fonte: Azevedo (2003)

Para realizar a translação de um ponto *P*, em um plano cartesiano bidimensional, basta realizar uma soma da coordenada atual [*x y*] por um escalar [*Tx Ty*], assim cada ponto pode ser movido por *Tx* unidades em relação ao eixo x e *Ty* unidades em relação ao eixo y. Desta forma, a nova posição do ponto *P* será obtida a partir da equação abaixo:

$$x' = x + Tx$$

$$y' = y + Ty$$

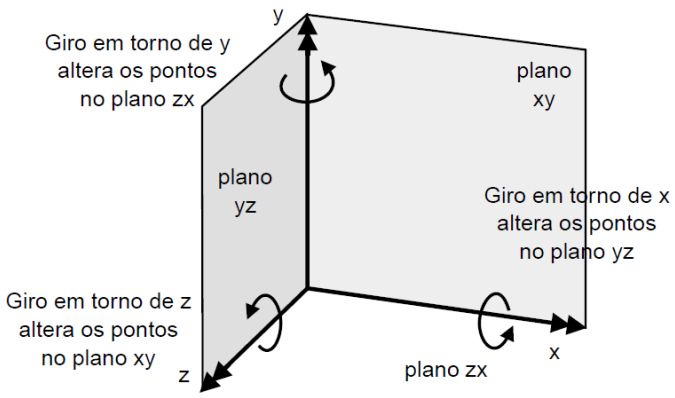
Esta equação pode ser reescrita da seguinte forma: $[x'\ y'] = [x\ y] + [Tx\ Ty]$. Além disso, ela pode ser estendida para espaços tridimensionais, utilizando coordenadas 3D (*x*, *y*, *z*). De forma análoga, esta é a equação de translação em espaços tridimensionais: $[x'\ y'\ z'] = [x\ y\ z]\ +\ [Tx\ Ty\ Tz]$.

2.2.4.2 Transformação de rotação

A rotação de objetos possibilita que a sua visualização em diferentes posições e ângulos, essa operação pode ser realizada de duas formas: uma delas seria girar o objeto no espaço por um certo ângulo; a outra forma seria rotacionar o próprio espaço ao redor do objeto por este mesmo ângulo. Uma das formas mais simples de realizar esta transformação é a partir de rotações individuais ao redor de cada eixo, usando os denominados ângulos Euler.

O ângulo de Euler descreve a rotação de um ponto, ou conjunto de pontos, em um plano 2D a partir do giro ao redor do vetor normal a este plano, isto é, um vetor perpendicular a este plano 2D. Exemplificando, um vetor no eixo *X* seria normal ao plano *YZ*, então um giro ao redor deste eixo rotacionaria os pontos no plano *YZ*, como está sendo demonstrado na figura 7.

Figura 7 – Ângulos de Euler



Fonte: Azevedo (2003)

2.2.4.3 Transformação de escala

A operação de escala, ou escalonamento, consiste em alterar o tamanho do objeto, por meio da multiplicação dos valores das coordenadas (*x*, *y*) de cada ponto que compõem esse objeto por um fator de escala *S*. A operação de escala é descrita com a seguinte equação:

$$x' = x \times S_x$$

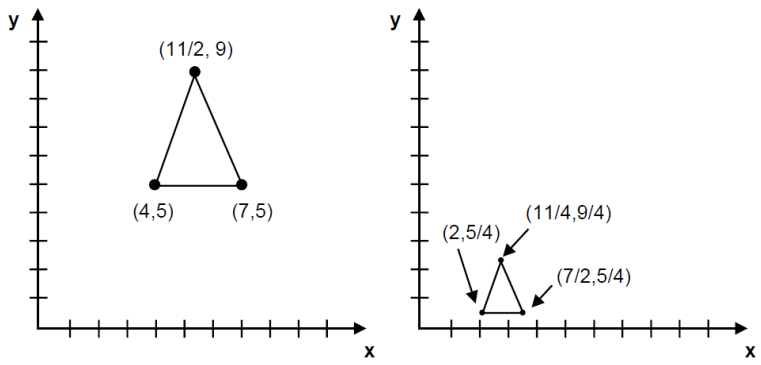
$$y' = y \times S_y$$

Essa operação também pode ser descrita na forma matricial:

$$[x'\ y'] = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}$$

A figura 8 demonstra o escalonamento de três pontos de um triângulo, com um fator de escala igual à ½ no eixo horizontal (eixo x), e outro fator de escala igual à ¼ no eixo vertical (eixo y). Percebe-se que é possível obter o mesmo triângulo da esquerda, se aplicarmos um escalonamento no triângulo da direita por um fator de escala 2 no eixo horizontal e um fator de escala 4 no eixo vertical. Em outras palavras, essa transformação é reversível.

Figura 8 – Escalonamento em objetos



Fonte: Azevedo (2003)

O escalonamento de um ponto no espaço tridimensional pode ser realizado de forma análoga à sua aplicação em 3D, com a multiplicação de 3 fatores de escala, um para cada componente da coordenada. Esta operação pode ser descrita como uma

multiplicação das coordenadas do ponto por uma matriz diagonal, cujo os valores não nulos são os fatores de escala.

$$[x'\ y'\ z'] = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix} = [xS_x\ \ yS_y\ \ zS_z]$$

2.2.5 Câmera Virtual

Na computação gráfica, o processo de projeção de cenários 3D em planos 2D, para síntese de imagens computacionais é comparado com o processo de fotografar um cenário real, a partir das lentes de uma câmera fotográfica. O processo fotográfico de um ambiente costuma ser descrito com os seguintes passos: posicionamento da máquina fotográfica; ajuste da orientação desta máquina, para decidir qual porção do ambiente será captada pelas lentes; ajustes no campo de visão da lente, através da regulagem da câmera, como controle de zoom, distância focal ou profundidade de campo.

Um modelo de câmera virtual, tanto de projeção em perspectiva quanto de projeção paralela, é definido pela posição da câmera, o seu ponto focal, o campo de visão, o plano de imagem, o vetor que indica o lado de cima da cena 3D, denominado *view up*, e a direção de projeção (*Look Direction*), que consiste em um vetor, o qual vai da posição da câmera até o seu ponto focal, indicando a direção que a câmera está apontada.

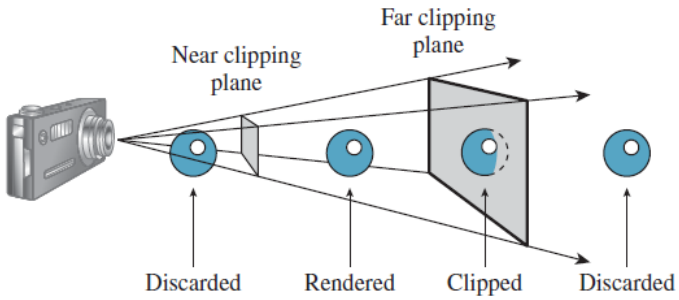
2.2.5.1 Camera Frustum

A especificação de visão da câmera, a partir dos elementos apresentados anteriormente, descrevem um volume de visão, o qual corresponde ao termo *Camera Frustum*, em que objetos dentro deste volume serão exibidos na imagem, enquanto que os objetos fora do volume não serão. Esta operação é conhecida como *Frustum Culling* e é bastante utilizada em jogos para a redução do tempo de renderização, aumentando a taxa de quadros.

Este volume é interceptado por 2 planos, os quais limitam o alcance de visão da câmera, conhecidos como planos de recorte (*clipping planes*). Um dos planos, denominado *Near clipping plane*, faz com que objetos os quais estejam em uma distância

menor do que a distância entre a câmera e este plano não interfiram na cena, isso quer dizer que objetos que estejam muito próximo ou atrás da câmera serão removidos. O outro plano é o *Far clipping plane* que remove objetos os quais estejam mais distantes do que este plano, em relação à câmera. Como é mostrado na figura 9.

Figura 9 – *Camera Frustum* em projeções perspectiva

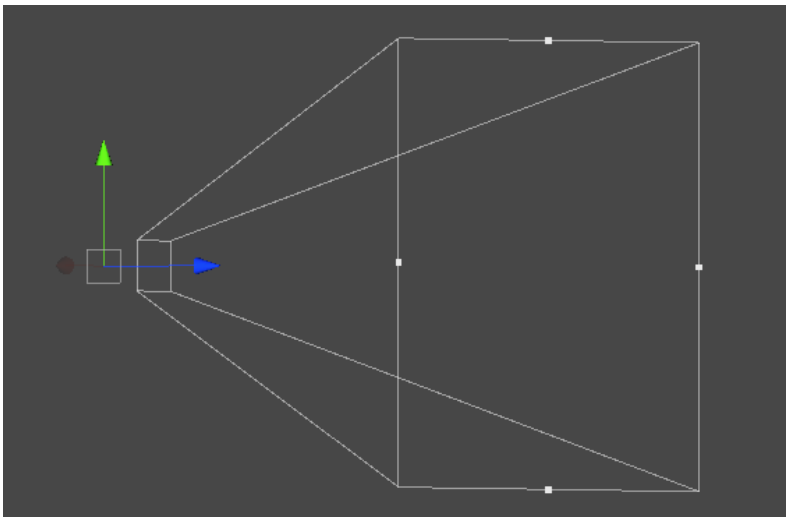


Fonte: Hughes et al. (2014)

2.2.5.2 Projeção perspectiva

As câmeras com projeções perspectiva se assemelham às câmeras físicas, possuindo um *frustum* no formato de pirâmide quadrangular, cuja base seria o *far plane* e o cume é cortado pelo *near plane*, conforme é mostrado na figura 10. Os tamanhos dos planos de recorte são definidos pelo valor do campo de visão da câmera e a proporção de aspecto (*aspect ratio*) da imagem à qual será projetada, que é calculado pela razão entre largura e altura ($w \div h$).

Figura 10 – *Camera Frustum* em perspectiva (*Unity*)



Fonte: Concebido pelo autor

As imagens obtidas, a partir deste tipo de câmera, apresentam o comportamento de perspectiva, que havia sido descrito na seção 2.2.2. Desta forma, objetos mais próximos possuem um tamanho maior e objetos distantes possuem um tamanho menor, o que ocorre devido a sua distância relativa ao observador (a câmera na cena). Um exemplo de resultado obtido com projeção perspectiva pode ser visto na figura 11.

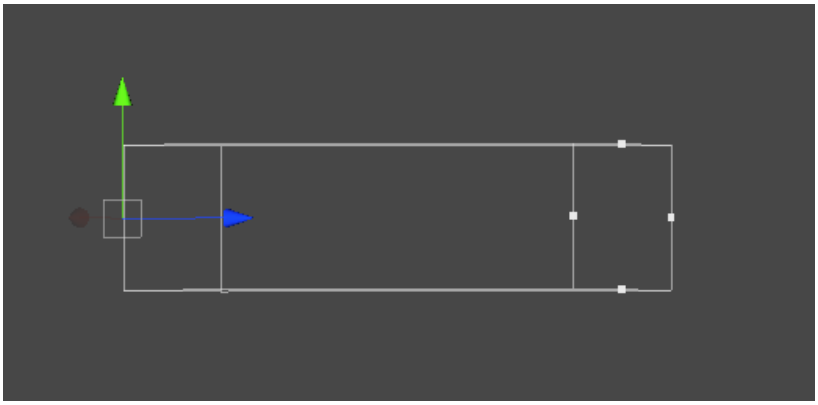
Figura 11 – Imagem em projeção perspectiva

Fonte: Concebido pelo autor

2.2.5.3 Projeção ortográfica

A câmera de projeção ortográfica (ou paralela) possui o *frustum* com um volume correspondente à um paralelepípedo, como mostrado na figura 12, fazendo a imagem resultante ser composta por linhas paralelas. Os tamanhos dos planos de recorte são definidos pela proporção do aspecto da tela e pela largura ou altura de um dos planos, a depender da aplicação, contanto que se mantenha a proporção do *aspect ratio*, para evitar distorção na imagem.

Figura 12 – *Camera Frustum* de projeção paralela (*Unity*)



Fonte: Concebido pelo autor

A imagem resultante apresenta os objetos no cenário em sua escala original, ou seja, os elementos da cena mantêm um tamanho constante, independente de estar mais perto ou longe do observador. Porém, ainda somos capazes de identificar, mesmo de forma mais imprecisa, que objetos estão na frente de outros através da oclusão, como mostra a figura 13. Importante ressaltar que esta imagem foi obtida por uma câmera virtual com a mesma posição e direção de projeção da câmera responsável pela figura 11, em outras palavras, são imagens correspondentes nos dois tipos de projeção.

Figura 13 – Imagem em projeção ortográfica

Fonte: Concebido pelo autor

2.2.6 Iluminação

A iluminação é um dos aspectos fundamentais para a composição de uma cena, sem ela, não seríamos capazes de visualizar coisa alguma no cenário virtual, uma vez que obteríamos uma imagem toda escura.

Segundo Azevedo (2003), modelos completos e abrangentes podem ser muito complexos ou até inexistentes. O que leva programadores a implementarem modelos de iluminação simplificados, os quais muitas vezes não possuem fundamento teórico no comportamento físico real, mas que na prática, apresentam bons resultados e são amplamente utilizados na computação gráfica.

Nos modelos digitais, os objetos podem ser emissores de luz, que correspondem as fontes de luz (lâmpadas, sol, fogo), ou refletores de luz, que recebem a luz e a reflete, evidenciando a sua coloração e brilho. Algumas propriedades dos emissores são a sua intensidade e frequência, enquanto que os refletores possuem propriedades sobre o material de sua superfície, como cor e polimento.

2.2.6.1 Luz direcional

A luz direcional (*Directional Light*) corresponde a uma fonte luminosa, que projeta raios de luz em uma única direção, isto é, raios paralelos de luz. A luz direcional não possui uma posição de origem identificável e não sofre variação de luminosidade em relação a distância dos objetos à fonte de luz, desta forma, a intensidade da luz é constante para objetos próximos ou distantes. Este tipo de luz é utilizado para representar um emissor natural do ambiente, como o sol ou a lua, simulando uma fonte de luz gigante que está à uma distância muito grande do cenário, conforme apresentado na figura 14.

Figura 14 – Representação de luz direcional

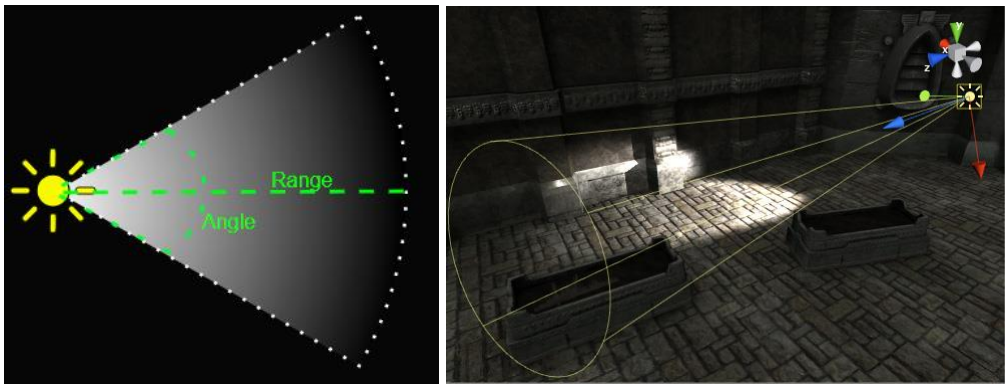
Fonte: Documentação da Unity[4]

2.2.6.2 Luz refletora

As luzes refletoras (*Spot Light*) possuem um alcance e angulação que determinam a região (em forma de cone), a qual será iluminada pela fonte de luz. Este tipo de luz é utilizado como luz artificial, para representar uma lanterna ou farol, por exemplo, conforme indica a figura 15.

[4] Disponível em: https://docs.unity3d.com/Manual/Lighting.html

Figura 15 – Representação da luz refletora

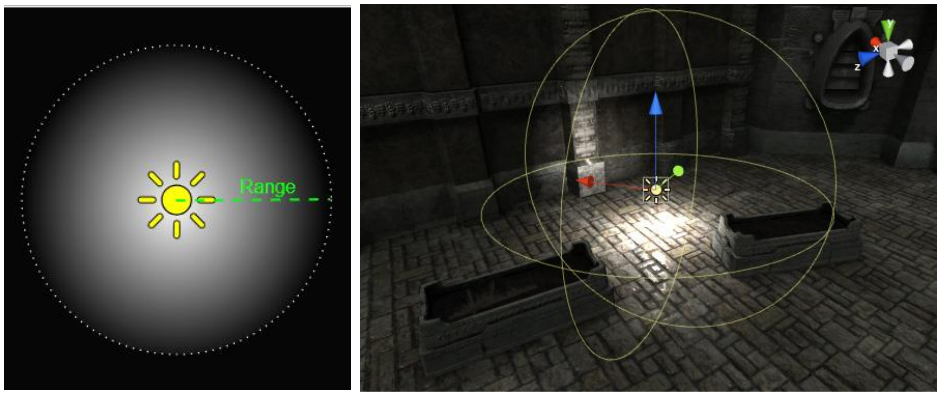


Fonte: Documentação da Unity

2.2.6.3 Luz pontual

A luz pontual (*Point Light*) está localizada em uma posição espacial específica e emite luz para todas as direções, é comum ser usada como fonte de luz em lâmpadas de cenários virtuais. Este tipo de luz pode ser interpretado como se fosse uma esfera, em que objetos próximos ficam mais iluminados e os objetos mais distante vão sendo menos afetados por esta luz, a uma proporção inversa ao quadrado da distância entre esse objeto e a fonte de luz, da mesma forma como a luz se comporta no mundo real, ver figura 16.

Figura 16 – Representação da luz pontual



Fonte: Documentação da Unity

2.2.6.4 Sombras projetadas

As sombras são elementos bastante importantes para aumentar o realismo das imagens. Normalmente, o sombreamento em objetos possui 2 tipos de regiões, a região

de sombra, onde não há intensidade luminosa e a região de penumbra, onde há uma intensidade luminosa que varia de zero até a intensidade da luz.

A sombra projetada é um algoritmo geométrico que consiste em projetar sombras em um plano de superfície, a partir do modelo do objeto. Esse método pode ser eficiente na projeção de sombras em superfícies planas, como piso ou parede, no entanto pode se tornar complexo, à medida em que o número de superfícies aumenta.

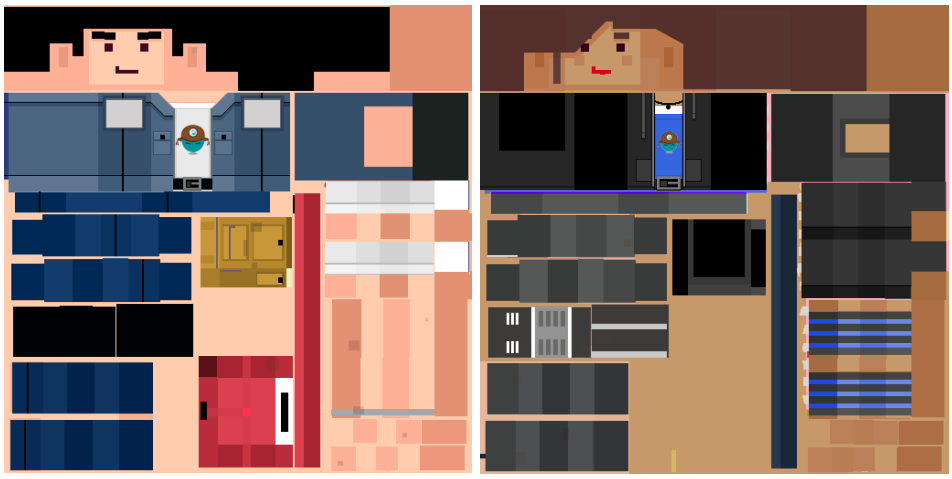
2.2.7 Texturas

Quando nos referimos às texturas, comumente pensamos em propriedades de materiais, como algo liso, rugoso, áspero ou brilhoso. Na computação gráfica, esse termo se aplica normalmente aos mapas de texturas, que consistem em imagens (*bitmaps*) aplicadas em modelos 3D, como se fossem uma espécie de pele do objeto, adicionando detalhes à aparência do modelo. Esta técnica foi inicialmente proposta por Catmull, em 1974, e pode ser usada em cenas complexas com um baixo custo computacional.

Esses mapas são compostos por 3 coordenadas, denominadas U, V e W, correspondendo as três letras do alfabeto que precedem X, Y e Z, que são as coordenadas dos objetos na cena. Neste sistema de coordenadas, o U corresponde ao eixo horizontal (equivalente ao X), o V corresponde ao eixo vertical (equivalente ao eixo Y) e o W corresponde ao eixo de profundidade (equivalente ao eixo Z).

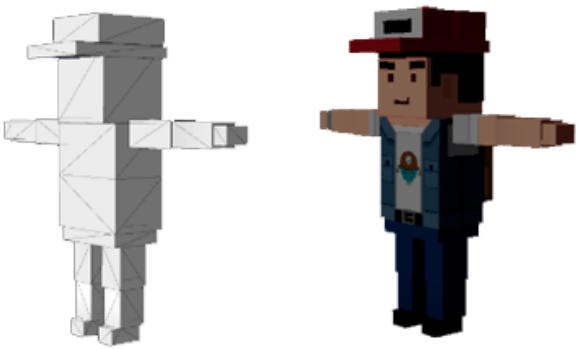
A textura deve ser deformada de acordo com a superfície do objeto, a partir do mapeamento das coordenadas da textura na superfície. Segundo Hughes et al. (2014), a atribuição das coordenadas de textura ocorre nos vértices da malha e, posteriormente, são realizadas interpolações lineares para expandir parte da imagem e ocupar o interior de cada face da malha. As figuras 17 e 18 demonstram a aplicação de texturas na malha do personagem do jogo “Recruta Social”.

Figura 17 – Texturas de personagem



Fonte: Concebido pelo autor

Figura 18 – Aplicação de texturas no modelo 3D

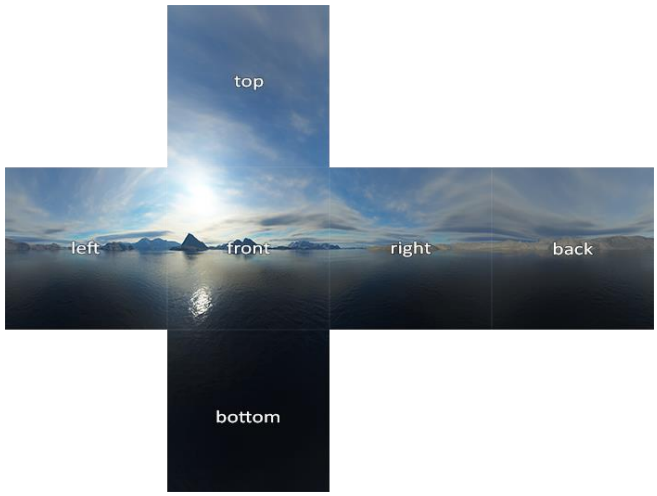


Fonte: Concebido pelo autor

2.2.7.1 Mapa de reflexão

O mapa de reflexão, também conhecido como *Environment Mapping*, é um tipo de mapeamento o qual projeta elementos que compõem a cena na superfície do objeto. Uma das formas de aplicar esse mapeamento é através de um objeto denominado *cubemap*, que consistem em um cubo onde cada uma das suas faces possuem uma imagem de textura, uma para cada direção da face, contendo informações de elementos que compõem o cenário, conforme é mostrado na figura 19.

Figura 19 – Representação de *cubmap*



Fonte: Learn Opengl[5]

Desta forma, para cada vértice do polígono, é calculado um vetor de reflexão. A outra forma de criar mapas de reflexão é gerando a imagem, a partir de uma esfera que reflete o ambiente. Como é mostrado na figura 20.

Figura 20 – *Reflection Probe* (*Unity*)

Fonte: Concebido pelo autor

A *Unity Engine* aplica reflexão em materiais refletivos, através de *cubemaps*, cuja as faces armazenam mapas de texturas que são gerados, a partir das imagens obtidas de

[5] Disponível em: https://learnopengl.com/Advanced-OpenGL/Cubemaps

uma esfera refletora, que visualiza os arredores do ambiente em todas as direções, como o demonstrado na figura 20, que foi obtida na própria *engine*.

2.2.7.2 Light Map

Os *light maps* consistem em mapas de texturas que armazenam informações a respeito da iluminação pré-calculada em objetos na cena, reduzindo desta forma os cálculos de iluminação complexa nos objetos em tempo real, aumentando a velocidade da renderização de objetos na cena.

Em outras palavras, realiza-se um cálculo prévio de distribuição da luz sobre os objetos e armazena esses resultados na forma de uma textura, que será aplicada em objetos 3D, ao invés de calcular esse efeito de luz no objeto em tempo real. Essa técnica possui algumas restrições, como mudança de posição de objetos ou fontes de luz que podem exigir o recalculo dos efeitos de iluminação e exigir a correção das coordenadas do mapeamento, podendo causar um grande impacto na *performance*. Além disso, esta técnica exige um maior uso de memória, para armazenar os *light maps*. A figura 21 demonstra a aplicação de *light map* na textura de um objeto.

Figura 21 – Lights maps

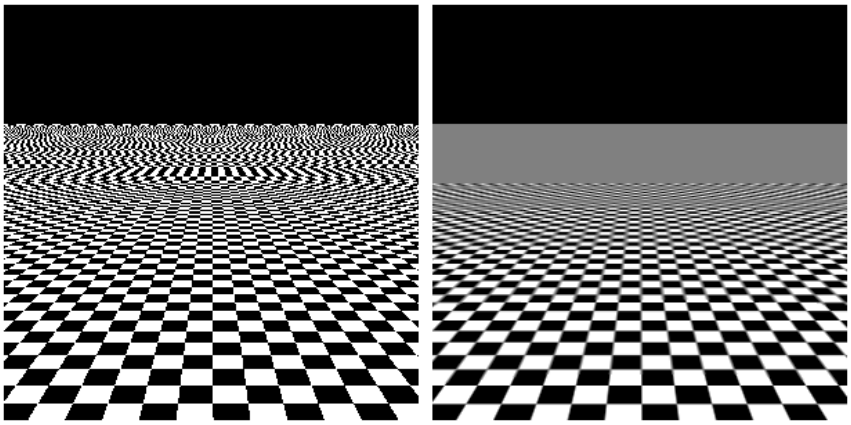
Fonte: Azevedo (2003)

2.2.7.3 Mipmap

Mip-mapping é uma técnica desenvolvida por Lance Williams para resolver um problema de distorção, conhecido como padrão de Moire. Esta distorção ocorre quando se utiliza uma única textura em objetos a uma distância muito grande do observador, como por exemplo o plano com textura de xadrez mostrado na imagem a esquerda da

figura 22, onde a textura vista a uma curta distância é apresentada sem distorção, e à medida que a textura se distancia, ocorre a deformação.

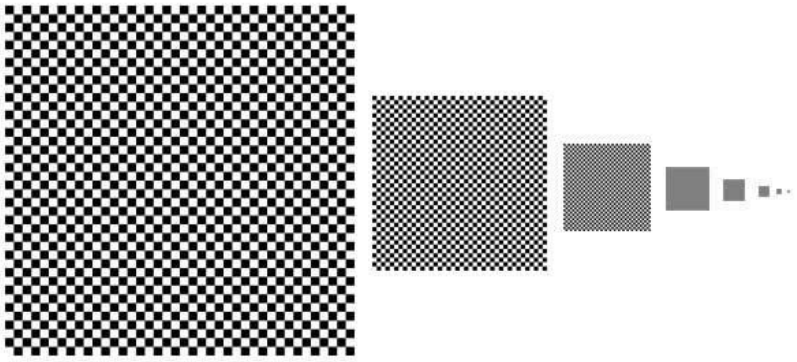
Figura 22 – Aplicação de *mipmap*



Fonte: Blog “textureingraphics”[6]

Para mitigar esse problema, é utilizada a técnica do *mip-mapping*, que consiste em utilizar mais de uma imagem como textura, em que cada imagem representa uma versão redimensionada da textura original, como pode ser observado na figura 23. Observando o padrão de decrescimento das texturas, percebe-se que a textura vai sendo reduzida pela metade de suas dimensões, isto se repete até o último *mipmap* possuir dimensão 1x1.

Figura 23 – Redimensionamento de texturas



Fonte: Blog “textureingraphics”

[6] Disponível em: https://textureingraphics.wordpress.com/what-is-texture-mapping/anti-aliasing-problem-and-mipmapping/

2.3 Desenvolvimento na Unity

Os motores de jogos (*game engines*), também conhecidos como motores gráficos, consistem em programas que possuem um conjunto de ferramentas e bibliotecas as quais são utilizadas para viabilizar, facilitar e impulsionar o desenvolvimento de jogos, tanto 2D como 3D, dependendo da *engine*.

Com esses programas, os desenvolvedores de jogos poderão ter um ponto de partida para iniciar o processo de desenvolvimento, sem a necessidade de implementar tudo do zero, como funções de renderização, simulações de física ou sistemas de animações. Permitindo desta forma, que equipes menores ou até indivíduo sejam capazes de desenvolver seus jogos, com menor risco, custo e tempo, garantindo o foco no que é mais importante; a produção do *game*.

Dentre as *engines* de jogos disponíveis atualmente, a *Unity 3D* se destaca por possuir várias ferramentas robustas, bem documentadas e de fácil aprendizagem, além de facilitar o desenvolvimento ágil de aplicações. Este motor é um dos líderes no mercado mundial, dando suporte a mais de 25 plataformas, dentre elas: *Android*, *iOS*, *macOS*, *Windows* e *Linux*. Além de possuir 5 tipos de licenças: *personal*, plus, pro, educacional e empresarial. Sendo a primeira licença, *personal*, gratuita. Devido a essas propriedades, foi utilizado a *Unity 3D* como *engine* para o desenvolvimento do jogo.

Alguns conceitos fundamentais e principais ferramentas e bibliotecas para o desenvolvimento com a Unity serão apresentadas e descritas a seguir, sendo elas: *Assets*; O editor da *Unity*; *GameObjects* e componentes; *Prefabs*; física na *Unity*; *Monobehaviour*; *Coroutine*; *Canvas*. Mais detalhes sobre estes conceitos e bibliotecas podem ser encontrados na documentação da *Unity*[7].

[7] Disponível em: https://docs.unity3d.com/ScriptReference/index.html

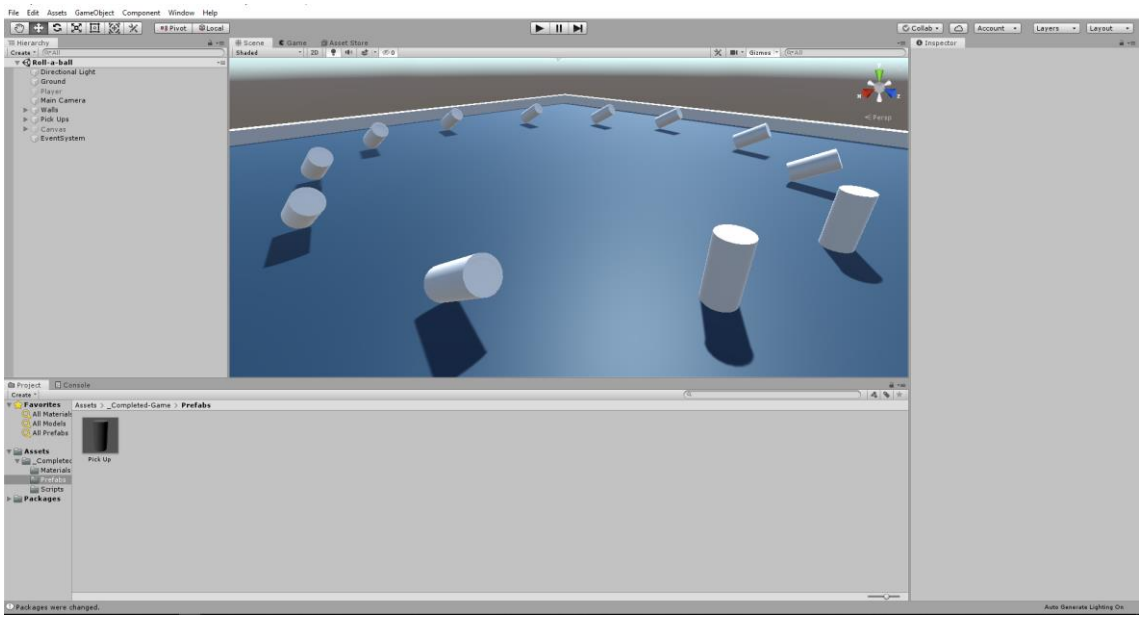
2.3.1 O que são assets?

O termo *assets* representa os recursos do projeto, isto é, arquivos necessários para o desenvolvimento do jogo. Podendo ser imagens, vídeos, modelos 3D, texturas, *scripts*, materiais e outros recursos que podem ser utilizados para a criação do jogo.

2.3.2 O editor da Unity

O editor consiste na interface principal da *Unity*, em que o desenvolvedor pode compor os cenários do jogo, manipular objetos do jogo, acessar os *assets* (recursos), como imagens, modelos 3D, áudios ou scripts (ver figura 24). Além de possuir o modo Play, que permite executar o jogo sem a necessidade de gerar uma aplicação executável, impulsionando o desenvolvimento e os testes de validação do jogo. Uma descrição mais detalhada do editor pode ser encontrada na documentação da *Unity*[8].

Figura 24 – Editor da *Unity*



Fonte: Concebido pelo autor

[8] Disponível em: https://docs.unity3d.com/Manual/EditorFeatures.html

2.3.3 Visão geral sobre GameObjects

Os *GameObjects* são os objetos fundamentais da *Unity*, eles podem representar qualquer objeto presente na cena do jogo, como o avatar do jogador, câmeras, objetos do ambiente, o próprio cenário e podem ser ativados ou desativados em qualquer momento do jogo. As funcionalidades desses objetos são definidas pelos componentes acoplados a ele. Em outras palavras, estes objetos funcionam como um receptáculo (*container*) de componentes, em que cada um deles adicionam uma lógica específica (ou comportamento) para o objeto. Dentro deste tópico, o termo *GameObject* poderá ser referenciado apenas como objeto.

2.3.3.1 O que são componentes?

Para Porter (2013), componentes são como pequenos elementos de uma grande máquina, responsável por realizar uma única tarefa de forma independente (idealmente) aos demais componentes, sem levar em consideração o funcionamento de todo o sistema. Esse conceito é utilizado em engenharia de *software* com reuso, o que indica que o componente pode ser reutilizado em diversos sistemas.

Um exemplo sobre componentes, apresentado pelo Porter (2013) é um controle de *video game*, que é composto por vários botões, e cada um exerce a sua própria função, de forma independente aos demais. Sendo assim, os botões são componentes do controle e funcionam de forma independente, de modo que se um botão está sendo pressionado, este não levará em consideração se algum outro está sendo pressionado ou não. Além disso, o próprio controle pode ser considerado um componente de uma plataforma, como um console ou computador, dessa forma, ele funcionará de modo independente do sistema ao qual está conectado, sem levar em consideração se há algum outro periférico conectado a plataforma.

No editor da *Unity* é possível adicionar, alterar ou remover componentes dos objetos presentes na cena da aplicação. Os *GameObjects* podem ter qualquer quantidade de componentes acoplados a ele, porém sempre haverá um componente que estará presente em todos esses objetos. Este componente é denominado *Transform* e é

responsável por determinar a posição espacial, escala e rotação deste objeto. Sem esse componente, o objeto não possuiria uma posição no espaço, nem escala ou rotação, por esse motivo, todo *GameObject* deve possuir o *Transform*.

2.3.3.2 Hierarquia de objetos

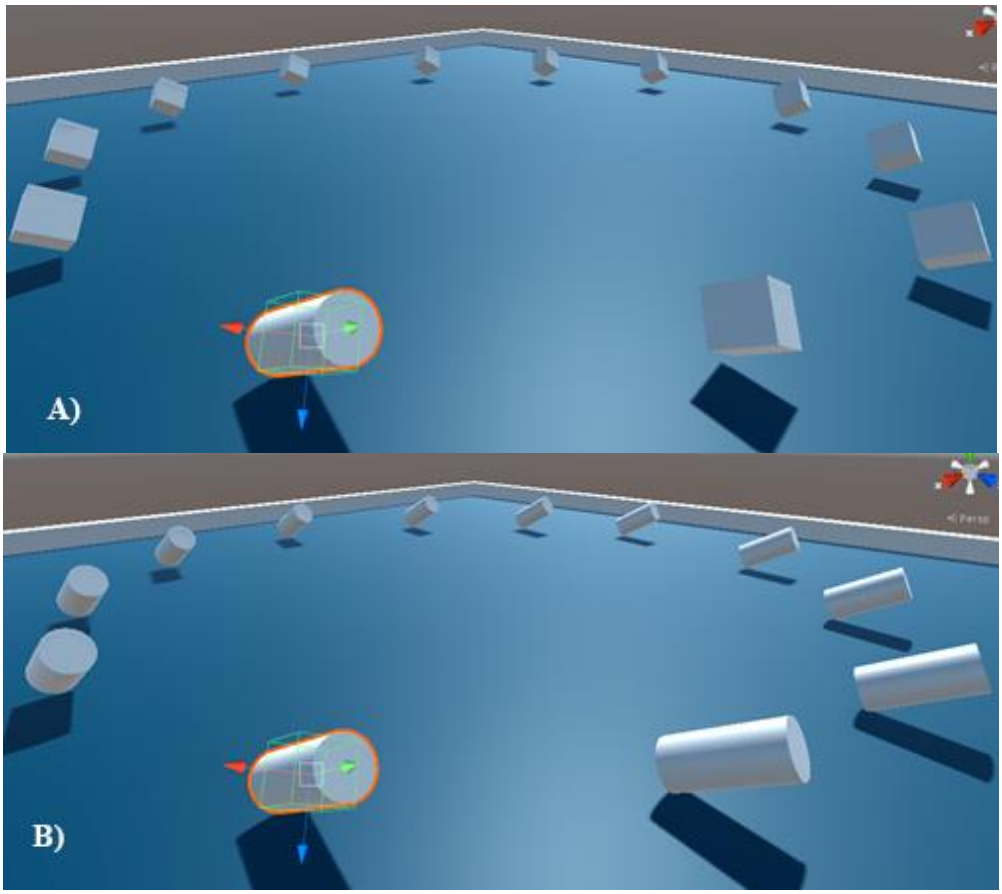
É possível estabelecer uma hierarquia de *GameObjects*, tendo um objeto como "pai", e este pode ter vários "filhos". Esta relação sempre será de 1 para muitos, no sentido que um "pai" pode ter vários "filhos", porém um "filho" só poder ter apenas um "pai". Este tipo de hierarquia estabelece uma relação de dependência dos filhos com o pai, de tal forma que alterações feitas ao *GameObject* no topo da hierarquia afetará todos os objetos em níveis mais baixos hierárquicos. Por exemplo, ao realizar operações de translação, rotação ou escalonamento em um *GameObject* que possui vários "filhos", esta mesma operação se propagará para todos eles.

2.3.4 Trabalhando com Prefabs

O sistema de *Prefabs* da *Unity* permite a criação, modificação e armazenamento de um *GameObject* completo, com todos os seus componentes, valores de propriedades e os seus "filhos" em um *asset* que poderá ser utilizado várias vezes em diversos cenários do projeto. Além disso, este recurso permite sincronizar os objetos, de forma que é possível alterar uma instância de objeto deste *Prefab* e aplicar essas mudanças as demais instâncias derivadas do mesmo *Prefab*, além de poder optar por reverter as alterações, retornando as configurações estabelecidas no *asset* ou manter estas modificações apenas nesta instância.

A figura 25(a) demonstra a utilização de *Prefabs* em uma cena de um jogo na *Unity*, em que todos esses cubos são *GameObjects* que foram criados a partir do mesmo *Prefab*, percebe-se que eles compartilham as mesmas propriedades. Então, foi realizado uma alteração em um desses cubos, transformando-o em um cilindro. Inicialmente, essa modificação só afetou apenas uma instância, na sequência, ela é aplicada ao *Prefab*, acarretando a alteração das demais instâncias do *Prefab*, que também se tornaram cilindros, como pode ser visto na figura 25(b).

Figura 25 – Demonstração de *Prefabs*



Fonte: Concebido pelo autor

2.3.5 Sistema de física da Unity

Em jogos eletrônicos é comum utilizar simulações físicas para ter mais realismo e adicionar novas mecânicas ao jogo. Alguns exemplos de aplicações da mecânica em jogos seriam: aplicações de força para movimentar objetos, como a força da gravidade; colisões entre objetos.

O motor de física da *Unity* fornece alguns componentes, tanto para cenas 2D quanto para 3D, que fazem com que os objetos, os quais os componentes estão vinculados, sejam afetados pela simulação de física da *engine*, sem a necessidade de implementar algum *script*. Desta forma, é possível fazer com que um objeto seja afetado pelas forças da gravidade, apenas com a adição de um componente (*Rigidbody*), ajustando as suas propriedades de massa e os valores da gravidade.

Mesmo que o projeto desenvolvido não tenha feito muito uso de física, dois componentes do sistema de física da *Unity* foram intensamente utilizados, são eles: *Rigidbody*, utilizado para implementar a movimentação do jogador e detecção de colisão; e os *Colliders* (colisores), que são utilizados para detectar colisão.

2.3.5.1 Rigidbody

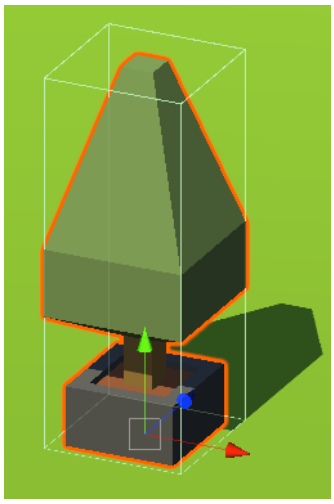
O *Rigidbody* é um componente do sistema de física da *Unity*, ele faz com que objetos sejam afetados pela física de forma realista, como reagir a uma força externa, cair devido a força da gravidade ou ser empurrado devido a uma colisão.

Além disso, esse componente é necessário para a detecção de colisão, ao mesmo tempo que é importante também ter o componente *Collider* para que a colisão ocorra. Deste modo, para o motor de física calcular uma colisão entre dois objetos, é necessário que pelo menos um destes tenham o componente *Rigidbody* e os dois objetos devem possuir o componente *Collider*. Isso serve tanto para os colisores comuns, isto é, objetos que não permitem ser atravessados por outros, quanto para os gatilhos (*Triggers*) que acionam alguns métodos, na ocorrência de algum evento.

2.3.5.2 Colliders

Os colisores são componentes que definem um modelo invisível o qual será levado em consideração para o cálculo de colisões. A figura 26 mostra um colisor, que corresponde ao paralelepípedo demarcado a árvore, percebe-se que este componente não precisa ter o mesmo formato da malha do modelo, basta ser apenas uma aproximação, utilizando colisores simples, o que já garante uma melhora na eficiência de processamento e um resultado semelhante ou idêntico no *gameplay*.

Figura 26 – Ilustração de colisor (na *Unity*)



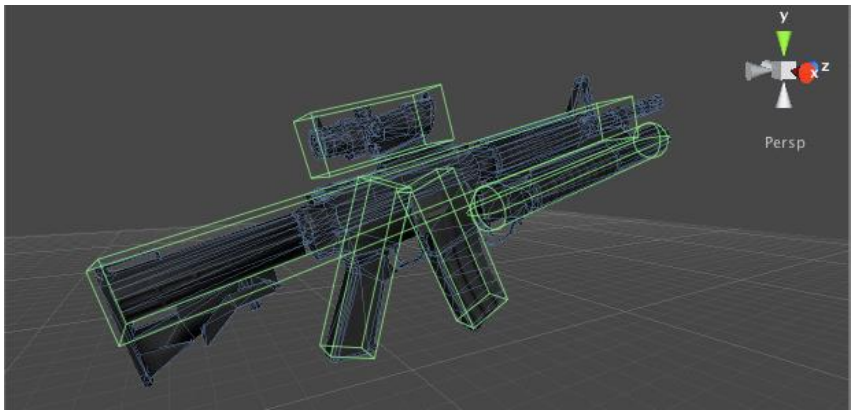
Fonte: Concebido pelo autor

Na *Unity*, esses colisores mais simples são denominados de colisores primitivos. Há diversos tipos deles, tanto para aplicações em 3D, como *Box Collider* (colisor em forma de caixa), *Sphere Collider* (colisor em forma de esfera) e *Capsule Collider* (colisor em forma de cápsula), quanto para 2D, como *Box Collider 2D* (colisor em forma de retângulo) e *Circle Collider 2D* (colisor em forma de círculo).

2.3.5.2.1 Composição de Colliders

É possível utilizar múltiplos colisores em um único objeto, como mostra a figura 27, permitindo adequar essa composição ao formato do modelo do objeto. Além disso, como alternativa, é possível utilizar o *Mesh Collider*, que cria colisor no formato da malha do objeto. Embora possa parecer conveniente, esta alternativa é mais ineficiente, uma vez que este tipo de colisor mais complexo aumenta a sobrecarga de processamento, sendo mais indicado utilizar uma composição de colisores primitivos.

Figura 27 – Composição de colisores



Fonte: Documentação da Unity[9]

2.3.5.2.2 Colliders comuns e Triggers

Não obstante as ideias apresentadas, há dois tipos de colisores na *Unity*, os colisores comuns (ou físicos) e os *triggers* (gatilhos). Os colisores físicos consistem em uma forma geométrica invisível que é impenetrável por outros colisores físicos, contanto que o objeto vinculado a um deles possua o componente *Rigidbody*, sendo comumente utilizados em objetos como chão, paredes, móveis, personagens ou qualquer outra coisa do cenário que não deva ser atravessada, dando o comportamento de um objeto sólido.

Por outro lado, os *Triggers* são formas geométricas invisíveis que são penetráveis, eles não se comportam como objetos sólidos e podem ser atravessados. Ao entrar em contato com objetos desse tipo de *Collider* é acionada algumas funções (*Callbacks*) em que é possível implementar lógicas específicas. Algumas dessas funções são: *OnTriggerEnter*, que é acionada quando um colisor entra em contato com este *trigger*; *OnTriggerStay*, acionado enquanto o colisor estiver dentro deste *Trigger*; e o *OnTriggerExit*, que é chamada no momento em que o outro colisor sai da área do *Trigger*.

[9] Disponível em: https://docs.unity3d.com/Manual/class-Rigidbody.html

2.3.6 Scripts na Unity

Embora a *Unity* forneça vários componentes nativos, que são versáteis, podendo ser utilizado em várias ocasiões, muitas vezes será necessário que o desenvolvedor implemente os seus próprios componentes, por meio de *Scripts*.

A implementação de lógicas e regras é algo fundamental para o desenvolvimento de jogos e é realizada por meio de programação de *Scripts*. Na *Unity*, a linguagem utilizada para o desenvolvimento é o *C#*, que consiste em uma linguagem de programação padrão da indústria, semelhante ao *Java* e *C++*. Alguns exemplos de *scripts* desenvolvidos no projeto são: fazer o personagem se movimentar, a partir do deslocamento do botão do analógico, isto é, fazer o personagem reagir as entradas do usuário; fazer o avatar interagir com outros personagens ou objetos do ambiente; realizar salvamento automático no jogo; entre outros.

2.3.6.1 A classe *Monobehaviour*

Monobehaviour é a classe base da qual todos os componentes derivados de *scripts* herdam, ou seja, todos eles são utilizados como componentes de objetos na *Unity* e devem herdar desta classe, reaproveitando os seus métodos e atributos bases.

Cada instância de objeto, que possuir determinado componente em *script* (*Monobehaviour*), terá a execução do código deste componente feita de forma paralela e independente. Por exemplo, se houver vários personagens que possuem um componente o qual realiza a movimentação pelo cenário, cada um deles vai se movimentar de maneira independente e ao mesmo tempo, conforme está implementado no código desse *script*.

Essa classe disponibiliza um conjunto de *API*s fundamentais para a programação de *scripts* na *Unity*. O termo *API* (*Application Program Interface*) consiste em um conjunto de interfaces que permitem acesso às bibliotecas de códigos as quais fornecem funcionalidades específicas, evitando, desta forma, que os programadores necessitem desenvolver tudo do zero.

Nessa seção será descrito alguns destes métodos imprescindíveis para o desenvolvimento de aplicações neste motor de jogo.

2.3.6.1.1 Os métodos Awake e Start

Esses métodos da classe *Monobehaviour* costumam ser executados na inicialização de cada cena da aplicação, isto é, a cada vez que uma nova cena é carregada, estes métodos são chamados e a lógica implementada dentro deles é executada.

As principais diferenças entre o método *Awake* e *Start* consistem na ordem e condição de execução destes métodos. O método *Awake* é executado quando a instância do objeto, que possui o *script* em questão, está sendo carregada, enquanto o método *Start* é executado no primeiro *frame* em que este *script* esteja habilitado, ou seja, o *Awake* é executado antes do *Start*. Além disso o método *Awake* é executado mesmo que o componente de *script* no qual ele está implementado esteja desabilitado, o mesmo não ocorre com o método *Start*, uma vez que é necessário que o componente do *script* esteja habilitado para que ele seja executado.

Com isso, cabe ao desenvolvedor saber qual dos dois métodos funcionaria melhor com a sua lógica, embora que, em muitos casos, possam apresentar o mesmo resultado. Porém, isso pode ser bem útil para manter uma certa sincronização de código, por exemplo, se houver algum trecho de código que deve ser executado antes de outros *scripts* executem o seu método *Start*.

2.3.6.1.2 Os métodos Update e FixedUpdate

O método *Update* é executado a cada *frame* do jogo, caso o componente (*Monobehaviour*) estiver habilitado, ele é executado continuamente durante o funcionamento do jogo. Então se o jogo estiver rodando a uma taxa de 60 quadros por segundos (*Frames per second – FPS*), esse método será executado 60 vezes por segundo, em cada uma de suas instâncias na cena do jogo.

Já o *FixedUpdate* é executado continuamente, mas de forma independente da taxa de quadros do jogo. Esse método é executado em passos de tempos regulares, que

normalmente não coincidem com *frame rate* do jogo, sendo muito utilizado para executar as *API*s de física da *Unity*, uma vez que estes cálculos devem (preferencialmente) ser executados de forma independente ao desempenho do jogo.

Para exemplificar, considera-se um jogo de corrida online em que os carros são movidos por meio de simulações de física, levando em conta a massa, aceleração, torque e a gravidade. Se a movimentação do veículo é feita no método *Update*, ela será dependente da taxa de quadros, então se um participante estiver jogando em um sistema melhor do que outros, ele terá uma vantagem, pois o seu carro será mais rápido devido ao fato do seu jogo estar funcionando a uma taxa de quadros maior. Por esse motivo, o comportamento de movimento do carro deve ser independente do *frame rate*.

2.3.6.1.3 Os métodos OnEnable e OnDisable

O método *OnEnable* é invocado sempre que o componente de *script* ou o próprio objeto são habilitados. Já o método *OnDisable* ocorre na situação inversa, quando este *script* ou objeto são desativados. Esses métodos podem ser úteis para adicionar uma lógica que deve ocorrer sempre que uma dessas condições aconteça.

2.3.6.1.4 Coroutines

Quando se faz uma chamada de um método, todo o trecho de código que estiver dentro do bloco desta função será executado de uma vez, sem poder ser interrompido. Uma vez que é preciso executar um trecho de código desta função, somente depois que determinada condição ocorra, como por exemplo, passado um certo tempo, executar um determinado trecho de código. Isso pode ser reproduzido em métodos tradicionais, utilizando instruções de condições (*if*) e laços de repetição (*while* ou *for*), simulando esse tipo de comportamento: “enquanto este evento não ocorrer, não execute este trecho de código”, realizando uma verificação repetidas vezes.

Uma alternativa aos métodos tradicionais é utilização *Coroutine*, que consiste em métodos os quais podem suspender a sua execução, a partir de um certo ponto, com a

instrução *yield*, e depois que certo evento ocorra, como passar determinada quantidade de tempo, a execução dele será retomada a partir do ponto em que parou.

Para complementar a explicação, um exemplo prático de um jogo de tiro será apresentado. Neste tipo de jogos é bastante comum existir uma restrição de tempo entre os disparos realizados pelo jogador, denominado taxa de tiro (*fire rate*). O segmento de código 1, demonstra uma das alternativas utilizando métodos tradicionais, como o método *Shoot*.

Código 1 – Implementação do *fire rate*

```
      // ...


      // Indica o tempo que deve decorrer entre os tiros.
      float fireRate = 0.25f;

      // Indica o próximo tempo que será permitido atirar
      float nextFire = 0.0f;


      // O Update é executado a cada frame
      private void Update() {

          // Verifica o tempo de execução (Time.time)
          // Se for maior do que tempo para realizar o outro tiro, é
possível disparar.
          if (Time.time > nextFire) {

              // Verifica se o jogador apertou o botão esquerdo do mouse
              if (Input.GetMouseButton(0)) {

                  // Realiza o disparo.
                  Instantiate(bullet);

                  // Indica o  tempo que será possível realizar o próximo
                  nextFire = Time.time + fireRate;
              }
          }
      }

      // ...
```

Fonte: Concebido pelo autor

O segmento de código 2 apresenta outra implementação para esta mesma lógica, utilizando *Coroutine*. Percebe-se que é possível obter o mesmo resultado utilizando qualquer uma das duas abordagens, porém a abordagem utilizando *Coroutine* reduz o

número de verificações, uma vez que a primeira estratégia realiza a verificação de tempo a cada *frame*. Agora, em uma situação em que há dezenas de personagens atirando, isso reduziria uma quantidade considerável de verificações a serem realizadas, melhorando a eficiência do código.

Código 2 – Implementação do *fire rate* utilizando *Coroutine*

```
      //...

      // Indica o tempo que deve decorrer entre os tiros.
      float fireRate = 0.25f;

      // Indica o se o jogador pode atirar ou não.
      bool canFire = true;

      // Update é invocado a cada frame
      void Update () {

          // Verifica se o jogador apertou o botão esquerdo do mouse
          if (Input.GetMouseButtonDown(0)) {
              // Verifica se o jogador pode atirar
              if (canFire) {
                  Instantiate(bullet); // Dispara o tiro, caso afirmativo
                  StartCoroutine (Cooldown ()); // Inicia o método
Coroutine
              }
          }
      }

      // A assinatura de um método Coroutine sempre retorna o tipo
IEnumerator.
      private IEnumerator Cooldown () {
          canFire = false; // Desabilita a capacidade do jogador atirar,
momentaneamente
          yield return new WaitForSeconds(fireRate); // Suspende a
execução deste método pelo valor de fireRate (0.25 segundos)
          canFire = true; // Habilita a capacidade de disparar
      }

      // ...
```

Fonte: Concebido pelo autor

2.3.7 Descrevendo a Unity UI

Interface de usuário em aplicações interativas, como jogos é algo crucial para a garantir uma boa experiência de usuário, já que elas definem os menus, botões e

navegações do jogo. Sendo necessárias para estabelecer uma comunicação e interação entre o usuário e a aplicação.

2.3.7.1 O componente *Canvas*

O *Canvas* é o elemento fundamental de uma interface gráfica de uma aplicação da *Unity*. Ele é responsável por desenhar os elementos da *UI* na tela do jogo, por esse motivo, todos os elementos da interface devem ser "filhos" de um *GameObject* com o componente *Canvas*. Ele é responsável por gerar as malhas de elementos da interface e desenhá-las na tela (renderizar), por meio da *GPU*.

Segundo o guia de boas práticas da *Unity*[10], os *Canvases* são responsáveis por gerar e combinar a geometria (malhas) dos componentes visuais da interface, de forma agrupá-los (*batching*), gerando comandos de renderização para o sistema gráfico da *Unity*. Quando algum dos elementos de um *canvas* é alterado, vai exigir que as geometrias sejam geradas novamente e reagrupá-las (*rabatching*). De acordo com Dundore (2017), esse processo ocorre basicamente em três passos:

- Recalcula os elementos de *layout*;
- Gera as malhas para os elementos habilitados, incluindo os transparentes (*alpha* igual a zero);
- Gera os materiais para agrupar as malhas.

2.3.7.2 Grapich Raycaster

O *Graphic Raycaster* é um componente do sistema de eventos da *Unity*, responsável por transformar a entrada do usuário, como cliques do *mouse* ou toques na tela em eventos, e despacha-os para os objetos da interface gráfica que são interativos (*Raycast Target*).

Mais especificamente, a tarefa do *Graphic Raycaster* consiste em coletar o conjunto de elementos da interface gráfica que são interativos e realizar uma checagem de intersecção entre o ponto de entrada (clique ou toque) e a área de cada um desses elementos, em outras palavras, verifica se esses objetos foram pressionados ou não. Caso

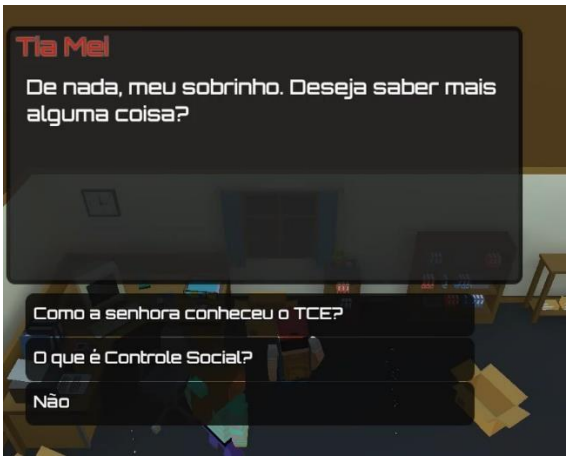
[10] Disponível em: https://unity3d.com/learn/tutorials/topics/best-practices/fundamentals-unity-ui

afirmativo, o *Graphic Raycaster* despacha um evento de *UI* para os objetos da interface lidarem com ele.

2.3.7.3 Componente *Layout*

O componente *Layout* controla o posicionamento e tamanho do conjunto de elementos que compõem uma determinada distribuição de membros de uma interface gráfica. Esse componente costuma ser utilizado quando se pretende construir uma distribuição complexa de elementos, como por exemplo, uma lista deslizante, contendo várias opções de escolha, como mostrado na figura 28.

Figura 28 – Exemplo de *layout* na *Unity*



Fonte: Concebido pelo autor

2.4 Princípios de otimizações em aplicações da Unity

Um dos aspectos de grande relevância no desenvolvimento de jogos eletrônicos está na otimização, em virtude da necessidade do programa rodar de forma fluída e rápida, de maneira que melhore a interação do jogador, pois não basta o jogo conter apenas um bom *design* e boas ideias, ele precisa ser executado de forma (satisfatoriamente) rápida, para poder funcionar em tempo real e garantir a satisfação do usuário.

A otimização possui como principal objetivo tornar o desempenho do jogo melhor sem sacrificar a sua qualidade, mantendo a aplicação mais interativa e agradável,

aumentando a sua disponibilidade, a fim de permitir uma maior quantidade de dispositivos capazes de rodar o programa.

Segundo Turner e Schell (2016), não há regras gerais para solucionar problemas de *performance* em jogos, pois eles costumam ser únicos para cada jogo, deste modo, nem todas as soluções que funcionaram para determinado jogo, vão funcionar em outros. Embora seja possível que jogos semelhantes compartilhem alguns problemas similares, o que pode facilitar a busca e implementação de soluções para este problema.

Otimizações em jogos exigem um melhor entendimento dos problemas encontrados e quais decisões devem ser tomadas para obter o melhor resultado de desempenho com o menor custo. Então, não basta apenas procurar soluções e seguir regras definidas por terceiros.

De acordo com Lira (2015), é preciso analisar o desempenho do jogo para descobrir, de forma precisa, onde se encontram os problemas. Evitando-se, dessa forma, o uso de suposições imprecisas que podem acarretar tomadas de decisões as quais demandam muito trabalho e resultam em poucos efeitos ou até efeitos negativos no desempenho, podendo comprometer diversos recursos lógicos ou visuais de forma desnecessária. Segundo Turner e Schell (2016), os primeiros passos para otimizar um jogo são:

- Analisar o desempenho do jogo (*Profile*);
- Avaliar os dados levantados pela análise (*Profile data*);
- Realizar ajustes e alterações, na tentativa de melhorar o desempenho;
- Avaliar o impacto das alterações;
- Passo extra: Avaliar o desempenho no pior dispositivo possível;
- Repetir todo o processo, até alcançar um desempenho satisfatório.

2.4.1 Métricas de *performance*

Em termos de otimização, há algumas métricas que devem ser levadas em consideração, como *Frame rate* e usos de recursos de *Hardware*, como *CPU*, memória e *GPU*.

2.4.1.1 Frame rate

Para um jogo funcionar de forma fluida é necessário que a taxa de quadros seja satisfatoriamente alta e estável.

Segundo Turner e Schell (2016), o fluxo de trabalho para a geração ocorre na seguinte ordem: primeiramente, a *CPU* atualiza os estados do jogo, verificando a entrada do usuário, executando *scripts* ou realizando cálculos específicos, como simulação de física; posteriormente, a *CPU* verifica o que precisa ser renderizado; essas informações são então repassadas à GPU, para serem renderizadas;

Quando essas tarefas são executadas em um tempo adequado, obtemos uma alta e estável taxa de quadros por segundo. Por outro lado, se alguma delas levar um tempo consideravelmente alto para serem concluídas, vai atrasar todo o processo de renderização da imagem, causando, consequentemente, a redução da taxa de quadros por segundos, comprometendo a estabilidade e o funcionamento do jogo.

2.4.1.2 Restrições de *Hardware*

Em relação ao *Hardware*, há diversos motivos que podem impactar no desempenho de um jogo, como: Problemas de temperatura, comprometendo o funcionamento do sistema; uso ineficiente de recursos do *Hardware*, como memória; ou utilização de sistemas com componentes muito limitados, com uma *CPU* ou *GPU* ultrapassadas.

Para Turner e Schell (2016), essas limitações de *Hardware* ocorrem em consequência de o desenvolvedor implementar soluções que são muito custosas, ou seja, que possam exigir muito processamento e muito trabalho para o sistema realizar. E dependendo do sistema, ele pode não ser capaz de executar estas tarefas em um tempo

hábil, comprometendo o desempenho do jogo, uma vez que os quadros vão levar mais tempo para serem processados.

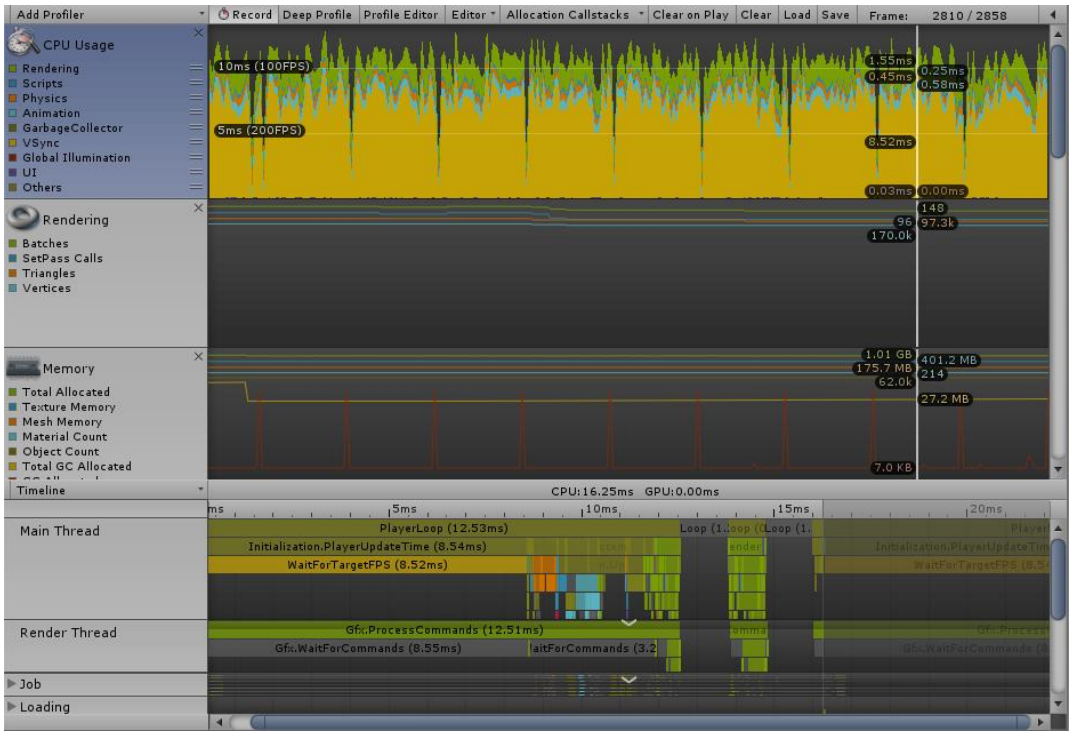
Quando o componente que está sendo sobrecarregado e comprometendo a performance do jogo é a *CPU*, diz-se que a aplicação é "*CPU Bound*", pois a aplicação está sendo limitada pela *CPU*. Isso ocorre quando a *CPU* não é capaz de realizar o processamento de suas operações a tempo, atrasando o funcionamento de toda a aplicação. De forma análoga, a aplicação pode estar sendo limitada pela *GPU*, sendo denominada "*GPU Bound*".

Para descobrir se a aplicação está sendo limitada por algum destes componentes, deve-se realizar uma análise do desempenho rodando na plataforma alvo, e verificar onde se encontra o fator que mais está limitando o desempenho. Existem ferramentas que ajudam o desenvolvedor a identificar quais processos estão causando mais impactos na *performance* e o uso dos recursos da máquina, como o *Unity Profiler*, que será descrito na próxima seção.

2.4.2 Unity Profiler

O *Unity Profiler* é uma ferramenta presente no editor da *Unity*, que mede o uso de recursos do *hardware* pelo jogo, durante o seu tempo de execução. Esse instrumento de medida demonstra, na forma de gráficos, o tempo de operação (em milissegundos) de cada processo sendo executado no jogo, o uso de memória, além de detalhes sobre o funcionamento de processos, como renderização e simulação de física. Conforme pode ser constatado na figura 29.

Figura 29 – *Unity Profiler*



Fonte: Concebido pelo autor

Segundo Gonzalez (2017), com essa ferramenta, é possível encontrar quais são os elementos do jogo que estão causando maior impacto no desempenho. Facilitando, dessa forma, o trabalho de otimização, uma vez que o desenvolvedor será capaz de identificar quais são os ajustes mais importante que precisarão ser realizados, quais scripts levam maior tempo para serem executados, o impacto da renderização, entre outras coisas.

É importante analisar a *performance* do jogo ao longo do tempo de sua execução, pois impactos de performance, conhecidos como "*Performance drops*", podem ser constantes ou esporádicos. Estas quedas de desempenho são representadas como "picos" nos gráficos do *Unity Profiler*, como mostra a figura 30.

Figura 30 – Gráficos com *Performance drops*

Fonte: Site do “Intentor”[11]

Se o problema encontrado foi em *scripts* feitos pelos desenvolvedores, então é necessário avaliar o código e ter um entendimento do problema que ele pode estar causando e como buscar solucioná-lo de forma eficiente e funcional. Caso seja algum processo da *Unity* que esteja causando problemas de *performance*, deve-se procurar soluções *online* ou alguém mais experiente com a *Unity* que possa ajudar.

2.4.3 Problemas comuns

Nesta seção, será discutido alguns pontos comuns no desenvolvimento de jogos que podem causar grande impacto na *performance*. Como aspectos visuais do jogo; *scripts* lentos; e o *Garbage Collection*.

2.4.3.1 Desempenho visual

Quando se fala em otimização da parte visual, no desenvolvimento de jogos, muitas vezes, é necessário pensar em estratégias de simplificação dos gráficos, que busque alcançar um efeito visual semelhante, porém menos custoso. No pior dos casos, pode ser necessário renunciar alguns modelos, materiais e efeitos em prol de um melhor

[11] Disponível em: http://intentor.com.br/otimizacao-de-jogos-com-unity-3d/

desempenho, desperdiçando esforços e trabalhos realizados. Então este tipo de otimização pode impactar tanto o *gameplay* quanto a qualidade visual do jogo.

2.4.3.2 Scripts lentos

Para Lira (2015), após identificar onde ocorre o *bottleneck* (gargalo), deve-se reescrever trechos de código, simplificar modelos e filtros, e otimizar recursos. Nessa parte a experiência pode ser o grande diferencial. Após o refatoramento, devem ser realizados testes e validação para verificar se o jogo está funcionando adequadamente, averiguando se foi obtido alguma melhora na *performance*. A repetição desse processo deve ser realizada até se alcançar uma taxa de quadros estável e satisfatória.

Segundo Turner e Schell (2016), algumas práticas podem ser levadas em consideração para amenizar o impacto de *scripts* lentos, são elas:

- Examinar a ineficiência de código, verificando se há chamada de funções custosas;
- Evitar execuções desnecessárias de código;
- Reduzir chamadas de funções custosas;
- Verificar o impacto em escala da execução desse *script*;

2.4.3.3 Heap Memory e Garbage Collection

Nos *scripts* há 2 tipos de alocação de memória, *Stack* e *Heap*, que ocorre quando uma variável é declarada, fazendo com que a aplicação requisite uma porção de memória. Enquanto essa variável estiver dentro do escopo de execução, essa porção permanecerá em uso (alocada).

De acordo com Teixeira (2018), a memória *Stack* consiste em um segmento de memória onde dados como as variáveis locais e as chamadas de funções são adicionados ou removidos, no modo *Last-in-First-Out* (*LIFO*), em que últimos elementos adicionados serão os primeiros a serem removidos da pilha. Por outro lado, a memória *Heap* consiste em outro segmento que não possui um tamanho constante e pode ser controlado pelo programador, sendo restringido pela capacidade física da memória. Esse segmento é utilizado: para armazenar estrutura de dados que não possuem um tamanho predefinido;

armazenar variáveis que persistam durante todo o tempo de execução da aplicação; ou quando é necessário alocar mais espaço do que o disponível.

Na memória *stack*, porções de memória que não estão sendo mais utilizadas retornam para a área de memória livre. Enquanto que na memória *Heap*, quando uma variável não está sendo mais usada, ela é definida como *garbage* e será removida pelo *Garbage collection*. Essa operação consiste em limpeza de memória, para reciclar memória *Heap* e pode se tornar um problema se ocorre com muita frequência na execução do código, uma vez que exige um processamento extra e o espaço alocado na memória *Heap* apenas expande, nunca sendo reduzido após a sua ampliação durante o tempo de execução.

Para reduzir o impacto do *Garbage Collection*, é preciso reduzir o número de alocações *Heap*. Uma alternativa é trocar (sempre que possível) alocações na memória *Heap* por alocações na memória *Stack*, utilizando estruturas de dados primitivas no lugar de modelos mais complexos.

É possível perceber o problema causado pelo excesso de operações do *Garbage Collection* quando ocorre travamentos momentâneos. Dessa forma, é preciso acompanhar quais funções estão realizando alocações na memória *Heap* durante a execução do jogo, verificando os dados de *GCAlloc* no *Profiler* da *Unity*.

3 PROPOSTA DO JOGO

3.1 Um pouco sobre o tribunal de contas

O Tribunal de contas do Estado da Paraíba (TCE-PB) é uma instituição que foi instaurada em 1970 com a responsabilidade de acompanhar e controlar as contas e os gastos do dinheiro público do Estado e dos municípios paraibanos.

O TCE-PB é responsável por examinar e acompanhar as contas públicas e operações financeiras realizadas por todas as Prefeituras Municipais do Estado da Paraíba, incluindo os poderes Executivo, Legislativo e órgãos da administração direta e indireta de todos os municípios do estado. Além disso, o TCE-PB acompanha a gestão estadual e municipal, fiscalizando e analisando os processos de prestações de contas, atos de admissão de pessoal, aposentadorias e pensões, licitações, contratos e convênios dos órgãos da administração estadual, entre secretarias; fundações; sociedades de economia mista; além das empresas públicas.

Para o Tribunal de Contas, é importante o apoio da sociedade nessa fiscalização, monitoramento e controle, pois exercer plenamente essas atividades sem tal apoio é impraticável, devido à grande diversidade de itens a serem fiscalizados, monitorados e controlados como também a grande área geográfica a cobrir. Sendo assim, é necessário estabelecer ações que fomentem na população o desejo de participar do controle social de forma correta e efetiva. Um dos caminhos possíveis, sem dúvida, é introduzir esse desejo e atitude durante a infância e adolescência. Nesse contexto, desenvolver um jogo digital surge como uma oportunidade para esse fim, daí o desenvolvimento do jogo “Recruta Social”.

3.2 A parceria entre o TCE e o LAViD

O projeto do jogo foi concebido a partir de uma parceria estabelecida entre o Tribunal de Contas do Estado da Paraíba e o Laboratório de Aplicação de Vídeo Digital

(LAViD), um dos principais laboratórios de desenvolvimento e pesquisa no centro de informática da Universidade Federal da Paraíba (UFPB).

O LAViD é responsável por desenvolver pesquisas e aplicações em diversas áreas na computação, como sistemas multimídia, sistemas embarcados, redes e sistemas distribuídos, além de desenvolver aplicações interativas para TV digital. Sendo uma grande referência, nacional e internacional, no mercado de tecnologia. Algumas das principais aplicações desenvolvidas pelo laboratório, são: Vlibras, um tradutor automático de português para libras; Pamin, que consiste em uma plataforma de divulgação de eventos culturais; GTAAAS, uma ferramenta de acessibilidade para conteúdo digital; entre outros. Para mais informações sobre o LAViD e seus projetos, visite o site do laboratório[12].

A parceria entre o LAViD e o TCE-PB concebeu um contrato que estipula o desenvolvimento de *Work Projects* (projetos de trabalho) para um processo de inovação tecnológica e inclusão digital proposto pelo tribunal de contas, denominado "TCE Interativo", que consiste na criação e implantação de soluções interativas para o TCE-PB.

3.3 Visão geral do "Recruta Social"

O "Recruta Social" é um dos projetos estipulados pelo contrato entre o TCE-PB e o LAViD. Esse projeto consiste em um *serious game* que busca ensinar o papel do Tribunal de Contas do Estado na vida pública dos cidadãos, além de tentar incentivar a prática de controle social para o jogador. O *game* apresenta diversas situações-problemas, em que o personagem está situado e deve buscar solucioná-las, através de interações com os personagens do ambiente, da exploração dos cenários e dos conhecimentos adquiridos sobre o controle social.

[12] Disponível em: http://lavid.ufpb.br/

3.3.1 Características do jogo

O “Recruta Social” é um *serious game* de aventura, sendo um gênero que dá grande ênfase na exploração de cenários e nos aspectos narrativos do jogo. O público alvo principal do jogo são crianças e adolescentes entre 10 a 15 anos de idade, a plataforma na qual estará disponível são *smpartphone* com sistema operacional *Android*, na versão *Kitkat* (4.4.4) ou superior, além de requerer que o aparelho tenha suporte à *API* gráfica *OpenGl* 3.0.

O principal foco do *gameplay* consiste na resolução de problemas enfrentados pela população, em forma de missões, causados pela má gestão pública e o uso irregular dos recursos públicos, por meio da interação do jogador com os cidadãos (*Non-Player Character* - NPC) e os objetos dispostos no cenário. Abaixo, seguem algumas características chaves em termos da *gameplay* do jogo:

- Câmera isométrica, com projeção perspectiva;
- Sistema de diálogos, com múltiplas escolhas;
- Mundo aberto com diversos objetos interativos, como *outdoors* e *NPC*s;
- *Minigames* que adicionam mais diversidade ao *gameplay* do jogo;
- Customização do avatar do jogador, através da loja do jogo.

3.3.2 Sinopse

O protagonista do jogo é um jovem adolescente que está se mudando em detrimento da reforma de sua escola. Ele passa a morar com a sua tia, que será o seu guia para conhecer a cidade e resolver problemas sociais encontrados por lá, ensinando a ele o papel do TCE-PB e incentivando o controle social.

A narrativa, conforme está descrito no documento de apresentação do jogo (2019), diz que: “Prestes a iniciar o ensino médio, você é convidado(a) pela sua tia Mei a passar um tempo na cidade dela, e continuar a estudar lá, dado que a escola de sua cidade natal

está em reforma há mais de um ano. Embora meio receoso(a), você segue com o plano e passa a morar com a tia Mei, descobrindo que essa escolha mostrará um lado proativo em você."

O documento ainda afirma que: durante o jogo, "Você enfrentará situações cotidianas nem um pouco confortáveis e perceberá que se conseguir convencer todos a se tornarem ativos na busca da resolução dos problemas, tudo será resolvido rapidamente."

3.3.3 Objetivos instrucionais

Alguns dos objetivos instrucionais definidos no documento de *design* (GDD) do jogo:

- Apresentar o papel do TCE;
- Explicar o que é controle social;
- Justificar que todo o cidadão tem o direito de reclamar por melhores serviços públicos;
- Demonstrar os passos necessários para realizar uma denúncia ao TCE;
- Apresentar o aplicativo "Nosso TCE";
- Validar que o controle social surte efeito na gestão pública.

4 CONCEPÇÃO DO JOGO

Neste capítulo, será discutido alguns aspectos sobre o *game design* e gerenciamento de projeto realizados no jogo “Recruta Social”, apresentando alguns elementos fundamentais para a sua construção e para alcançar os objetivos instrucionais, destacados na seção 3.3.3.

Os principais processos de tomada de decisão ocorriam através de reuniões, tanto presenciais como virtuais, realizadas semanalmente, durante todo o período do desenvolvimento do jogo. Nessas reuniões, discutiam-se sobre vários aspectos do jogo, como *gameplay*, narrativa, interface, narrativas e *level design*. Definindo características e tomando decisões que se alinhassem melhor com o interesse do TCE-PB, com isso, eram estabelecidas as tarefas que os desenvolvedores deveriam realizar e estipulava-se os prazos destas tarefas.

4.1 Construção do Gameplay

O *gameplay* consiste na experiência proporcionada pela interação entre o jogador e as mecânicas do jogo. É a partir disso que o “Recruta Social” procura atingir os seus objetivos instrucionais e tenta manter o jogador entretido. Sendo um dos principais desafios do desenvolvimento do jogo, o equilíbrio entre o aspecto sério e educativo com o fator diversão. Em alguns momentos do jogo, tenta-se fundir esses dois aspectos, proporcionando atividades lúdicas e educativas.

Algumas das principais atividades realizadas pelo jogador são:

- Exploração do ambiente de forma livre e arbitrária;
- Caminhar e correr pelos cenários;
- Realizar missões principais e secundárias;
- Diálogos com NPCs, com múltiplas escolhas;
- Fotografar determinados pontos do cenário;
- Interagir/Coletar objetos no cenário;
- Assistir vídeos educativos.

O quadro 3 apresenta algum dos possíveis comandos que podem ser realizados pelo jogador, bem como, a representação visual e descrição da ação desses comandos.

Quadro 3 – Principais comandos do “Recruta Social”

Controle	Figura	Ação
Analógico		Responsável pela movimentação do personagem.
Botão “Diálogo”		Botão que habilita o diálogo com os NPCs.
Botão “Correr”		Aumenta a velocidade da corrida do personagem.
Botão “Coletar”		Utilizado para coletar por determinados objetos dispostos no cenário.
Botão “Abrir”		Permite mudar de cenário.
Botão “Assistir”		Utilizado para assistir vídeos em determinados locais do cenário.
Botão “Tutorial”		Habilita o painel do tutorial do jogo, ensinando os comandos básicos para o jogador.

Botão “Configuração”		Abre o menu de configurações, podendo ajustar o volume do som, reiniciar e salvar o jogo.

Fonte: Concebido pelo autor

4.2 Idealização dos Minigames

Com o intuito de trazer mais diversidade ao *gameplay* e propostas divertidas ao jogo, a equipe de desenvolvedores elaborou sugestões e ideias para a implementação de minijogos (*minigames*), isto é, jogos curtos com mecânicas simples, que estão embutidos dentro do jogo principal. A ideia base era surpreender o jogador ao se deparar com uma variação de experiência (*gameplay*) e desafios, os quais não haviam sido explorados anteriormente no jogo, além de fornecer recompensas, podendo motivá-lo a continuar jogando.

A concepção dos *minigames* foi elaborada, a partir das reuniões de *brainstorm*, em que se discutia várias ideias de mecânicas e *gameplay*, que poderiam virem a ser implementadas e trazer uma experiência positiva ao usuário. No final do projeto, foram desenvolvidos cerca de 6 *minigames*, cada um com suas próprias mecânicas e desafios. Nos subtópicos seguintes serão apresentados e descritos 3 destes *minigames*, selecionados com o critério de relevância e originalidade para a narrativa.

4.2.1 Minigame da denúncia

Dentre os minijogos desenvolvidos no projeto, este é o que mais se alinha com os objetivos instrucionais do jogo. A sua principal proposta é ensinar o caminho para realizar uma denúncia ao TCE, apresentando o passo a passo de elaboração da denúncia.

O *gameplay* consiste no jogador tentando encontrar e coletar os objetivos, que representam um passo de uma denúncia, dentro de um labirinto e na ordem certa. Ele pode utilizar uma bússola, que aponta para a posição de um determinado alvo, ajudando-o a guiar e encontrar este objetivo. É preciso reiterar que os objetivos só podem ser

coletados em uma determinada ordem, a qual representa o caminho certo para realizar uma denúncia. Dito isso, é possível trocar o alvo no qual a bússola está apontando, cabendo ao jogador inferir qual é o próximo passo que deve ser feito para elaborar uma queixa ao TCE.

A movimentação do personagem neste minijogo é limitada e restrita, permitindo que ele se desloque apenas em determinadas posições (representadas como retângulos azuis no chão), assim ele pode se mover em apenas uma direção de cada vez. A cada deslocamento realizado, é gasto uma unidade na quantia de passos, portanto, a condição de derrota ocorre quando o jogador esgota toda a sua quantia de passos, sendo necessário reiniciar o *minigame* e tentar de novo. De acordo com a ilustração da figura 31.

Figura 31 – Minigame da denúncia

Fonte: Concebido pelo autor

Para concluir, o Quadro 4 apresenta, em ordem de coleta, as imagens e descrições dos objetivos que devem ser coletados.

Quadro 4 – Objetivos do minijogo da denúncia

Figura	Descrição
	Identificar um problema social.

	Registrar evidências deste problema.
	Elaborar o documento da denúncia.
	Registrar a denúncia ao TCE.
	Aguardar o resultado da denúncia e solução do problema.

Fonte: Concebido pelo autor

4.2.2 Minigame da d'água

O *minigame* da caixa d'água ocorre em uma missão secundária e é opcional, os aspectos da sua mecânica consiste em: fechar os buracos antes que o nível de água da caixa acabe, para isso o jogador deve pressionar os furos (com o dedo) até eles fecharem; além disso, é possível rotacionar a câmera ao redor da caixa d'água permitindo vedar os buracos em outras regiões da caixa; a condição de vitória ocorre, se o jogador não esgotar o nível de água até o tempo marcado no cronômetro acabar. Ver figura 32.

Figura 32 – Minigame da Caixa d'água

Fonte: Concebido pelo autor

4.2.3 Minigame da coleta de frutas

Este minijogo ocorre em um dado momento da história e possui as seguintes regras: o jogador deve coletar uma determinada quantidade de cada tipo de fruta (Pêra, maçã e banana) sem deixá-las cair; é possível movimentar o cesto em 3 posições (esquerda, direita e centro) para coletar as frutas; a condição de derrota ocorre quando o jogador deixa cair uma certa quantidade de frutas. Ver figura 33.

Figura 33 – Minigame das frutas

Fonte: Concebido pelo autor

4.3 Elaborando a narrativa

A narrativa desempenha um papel fundamental para que o jogo alcance os seus objetivos instrucionais, tendo em vista que este é o mecanismo utilizado para apresentar ao jogador as funções do TCE, explicar o conceito de controle social e estimular a sua prática, além de descrever os meios de realizar uma denúncia de um problema social. A trajetória do jogo é contada e desenvolvida, a partir dos diálogos com os *NPC*s.

Uma das exigências do cliente era que a função do TCE fosse apresentada nos primeiros momentos da história do *game*. Com isso, foram realizadas reuniões e sessões de *brainstorm*, em que cada participante fazia uma sugestão sobre a narrativa, além de elaborar os diálogos durante essas reuniões. Buscando, dessa forma, construir um enredo, em que toda a equipe colaborasse e se mantivesse informada e satisfeita com as decisões tomadas. Uma breve descrição do enredo está especificada na seção 3.3.2 e no Apêndice A é apresentado um exemplo dos diálogos elaborados em uma das reuniões do projeto.

4.4 Construção da interface

A interface de usuário (*User Interface* - UI) estabelece um meio que possibilita o usuário a interagir e se comunicar com a aplicação. Ela é necessária para garantir que o usuário possa ter algum controle sobre o jogo, podendo reagir as interações com os objetos e cenários do jogo.

Para elaborar a *UI* do jogo, foram realizadas algumas pesquisas com a finalidade de se obter referências artísticas e de *layout*, assim como foram criadas versões de protótipos, os quais permitiam validar as funcionalidades da mecânica e da interface, mesmo sem sua arte final, garantindo, desta forma, a interação com o jogo nas suas versões iniciais.

As artes da interface gráfica do jogo foram elaboradas seguindo um padrão de identidade visual, utilizando as cores da logomarca do TCE-PB, destacadas na figura 34. Além disso, foram utilizadas ferramentas de *design* gráfico, como programas de edição

de imagens (*Adobe Photoshop*) e programas de gráficos vetoriais (*Adobe Illustrator* e o *Inkspace*), para a criação dessas artes.

Figura 34 – Cores da identidade visual do jogo

Fonte: Concebido pelo autor

4.4.1 Interface principal

O *layout* da interface principal do jogo foi baseado na interface do *game* "*Assassin's Creed Identity*", por se tratar de um gênero de aventura, semelhante ao "Recruta Social", em relação aos aspectos como exploração e interação com os elementos do cenário, como objetos e NPCs.

Dito isso, é possível perceber, através das figura 35 e 36, algumas semelhanças na disposição dos elementos da interface, como os botões de comandos no canto inferior direito da tela, um minimapa disposto no canto superior direito da tela, além do botão com um ícone do personagem, possibilitando abrir a tela que apresenta detalhes dos *status* do jogador.

Figura 35 – Interface principal do jogo

Fonte: Concebido pelo autor

Figura 36 – Interface do "*Assassin's Creed: Identity*"

Fonte: Site "Lifewire"[13]

4.4.2 Interface dos diálogos

A interface dos diálogos passou por algumas reformulações, antes de chegar na sua aparência final. A figura 37 apresenta um modelo preliminar da interface do jogo, enquanto que a figura 38 demonstra um protótipo com a interface completamente

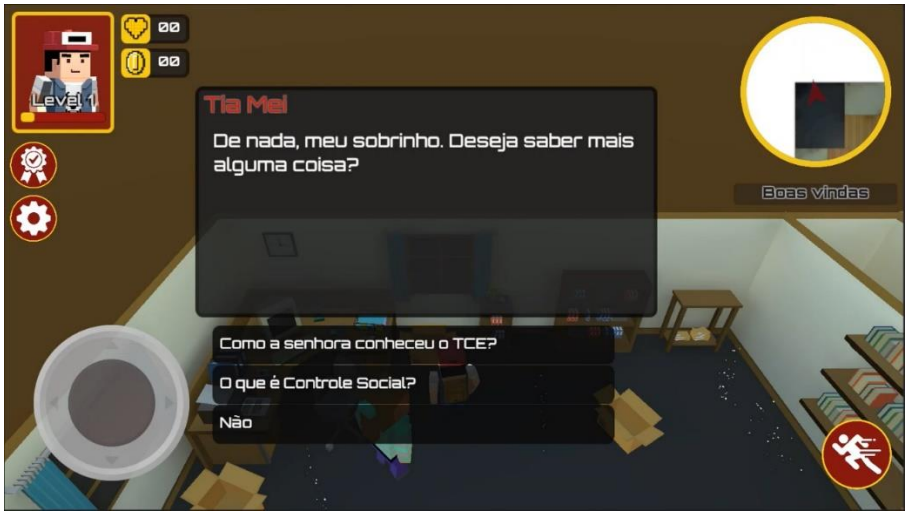
[13] Disponível em: https://www.lifewire.com/assassins-creed-identity-review-3896697

reformulada, o que era uma necessidade, devido ao fato da antiga *UI* comprometer a jogabilidade e a experiência proporcionada pelo jogo.

Figura 37 – Antiga interface

Fonte: Concebido pelo autor

Figura 38 – Interface reformulada



Fonte: Concebido pelo autor

A interface que apresenta os detalhes de avatar (figura 40) utilizou como inspiração a tela do jogo "*Hogwarts Mystery*" (figura 39). Na tela do avatar é possível verificar alguns estados do jogador, como: sua pontuação e quantia de dinheiro; além de

permitir acessar a loja, para poder customizar o personagem; apresentar um atalho para acessar os *minigames* do jogo; e possibilitar a visualização dos créditos e informações extras sobre o TCE-PB.

Figura 39 – Tela de avatar do "*Hogwarts Mystery*"

Fonte: Reddit[14]

Figura 40 – Tela de avatar do "Recruta Social"

Fonte: Concebido pelo autor

[14] Disponível em: https://www.reddit.com/r/harrypotter/comments/8ey7ar/

4.4.3 Personagem principal

Assim como na maioria dos jogos de aventura, o jogador realiza as suas ações e interações, através de um personagem controlável. Com isso, algumas decisões foram tomadas em relação a este personagem com o intuito de criar um vínculo mais próximo com o jogador. Dentre elas, algumas interessantes de citar são:

- O personagem pode ser do sexo masculino ou feminino (figura 41), podendo criar a satisfação de ambos os gêneros;
- O personagem é jovem, com a intenção de ser mais próximo ao público alvo;
- O personagem pode mudar de aparência, a partir da loja do jogo;

Figura 41 – Personagens principais

Fonte: Concebido pelo autor

4.4.4 Customização do personagem

Para dar um propósito às moedas do jogo e trazer conteúdo para a loja, foi definido uma customização do personagem jogável, permitindo que ele possa mudar a sua aparência em troca de moedas. Com isso, espera-se que esse ponto motive o jogador a buscar as recompensas (em moedas), podendo aumentar o seu grau de satisfação, além de prolongar o tempo de jogo.

Segundo Hussain e Griffiths (2008 apud King; Delfabbro; Griffiths, 2009, p. 97), a manipulação da identidade do avatar dentro do jogo digital pode ser uma parte importante da razão pela qual o jogador diverte-se com o *game*. Desta forma, a customização de personagens pode estimular o jogador a continuar jogando múltiplas vezes, utilizando diferentes aparências.

Sobre esse ponto de vista, foram desenvolvidas cerca de 14 roupagens (*Skins*) diferentes para o avatar, sendo 7 delas para a versão masculina do avatar, e as outras 7 para a versão feminina. De forma semelhante à criação da interface, foram utilizadas algumas referências para a elaboração das roupas do personagem, como demonstrado entre as figuras 42(a) e 42(b)**;** 43(a) e 43(b).

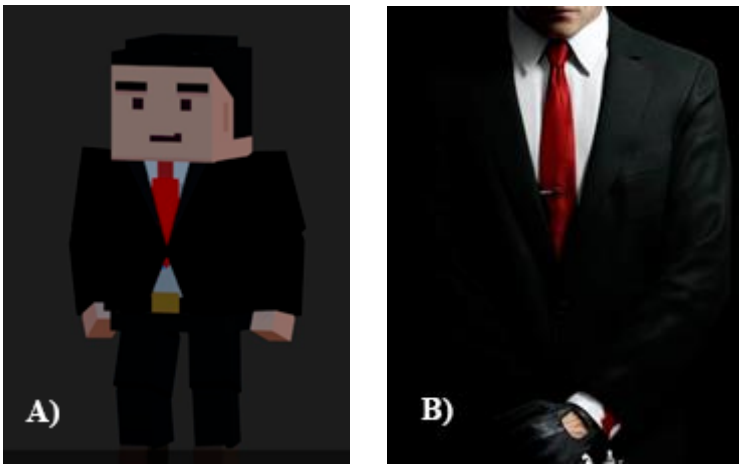
Figura 42 – Referência para roupagem feminina

Fonte: A) Concebido pelo autor.

B) *Website* elo7[15]

[15] Disponível em: https://www.elo7.com.br/quadro-the-last-of-us-06-25-x-35-cm/dp/9C7A0D

Figura 43 – Referência para roupagem masculina

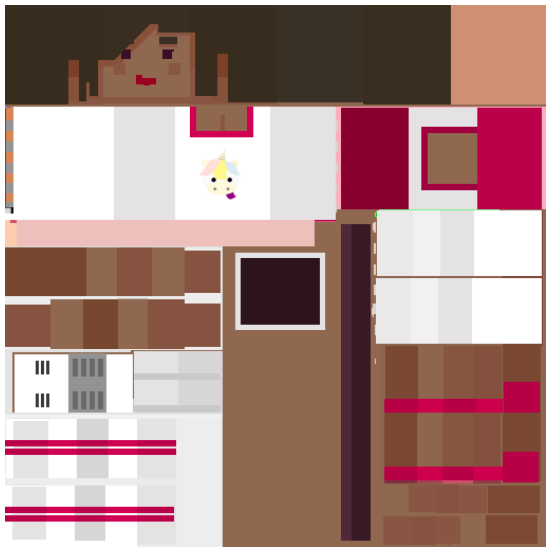


Fonte: A) Concebido pelo autor

B) Pinterest[16]

Estas roupagens criadas consistem em mapas de texturas, que foram elaboradas pelo autor deste trabalho de conclusão, utilizando *softwares* de edição de *bitmap*, mais especificamente, o *Adobe Photoshop*. A figura 44 demonstra um dos mapas de texturas criados para o avatar do personagem.

Figura 44 – Mapa de textura de uma *Skin*



Fonte: Concebido pelo autor

[16] Disponível em: https://www.pinterest.co.uk/pin/308496643218335071/

4.5 Level Design

No período de desenvolvimento do jogo, foram elaborados diversos cenários, alguns precisaram ser descartados, enquanto outros amplamente utilizados. A maior parte da construção desses cenários, a partir de alguns modelos criados pelo *Synty Studios*, disponíveis na loja de *assets* da *Unity* [17] . Desta forma, foi possível agilizar o desenvolvimento do projeto, dentro do prazo estimado, uma vez que o jogo foi desenvolvido por uma equipe pequena e sem muita ou nenhuma experiência com modelagem 3D. Contudo, o maior responsável pelo desenvolvimento do cenário final do jogo foi também o autor deste trabalho.

4.5.1 Cidade

A cidade, denominada "Magnólia" (apresentada na figura 45), é o cenário principal do jogo, onde o jogador vai realizar as principais atividades e passar a maior parte do tempo. Esse cenário possui vários *NPC*s, moedas, objetos coletáveis, missões principais e missões secundárias, além de ser o local de acesso dos demais cenários e os *minigames* do jogo.

[17] Disponível em: https://assetstore.unity.com/

Figura 45 – Cenário da cidade Magnólia

Fonte: Concebido pelo autor

4.5.2 Casa da tia Mei

Este cenário corresponde à casa do jogador, onde ele pode visitar e conversar com a tia Mei (personagem da história), que serve como instrutora para o jogador, ensinando-o sobre o papel do TCE e incentivando-o a exercer o controle social. Ver figura 46.

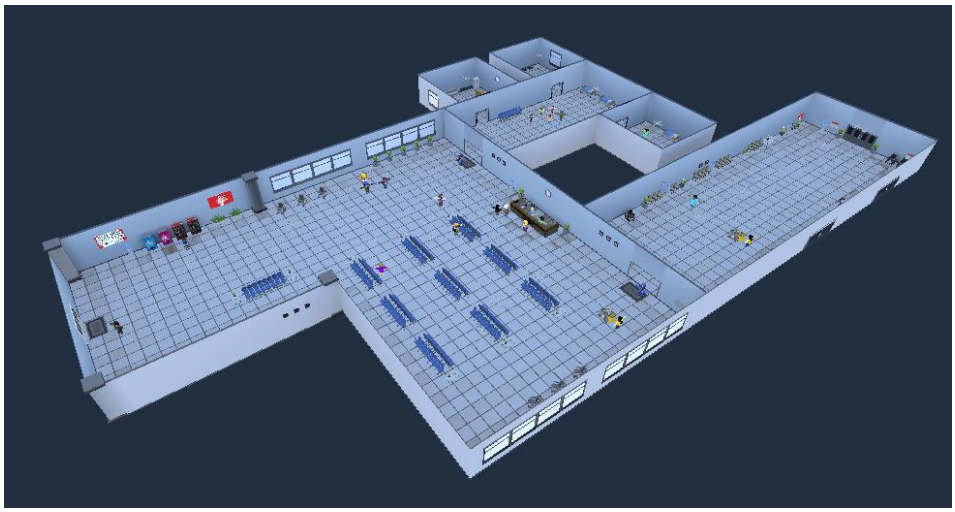
Figura 46 – Cenário da casa da Tia Mei

Fonte: Concebido pelo autor

4.5.3 Hospital

Esse cenário é acessado pelo jogador em um determinado momento da história do jogo, em que ele vai precisar ajudar na recuperação dos seus colegas de escola, com a ajuda da enfermeira e diretora do hospital. É o caminho para realizar um dos *minigames* principais do jogo. Ver figura 47.

Figura 47 – Cenário do hospital



Fonte: Concebido pelo autor

5 PROGRAMAÇÃO NO RECRUTA SOCIAL

Este capítulo vai descrever algumas das práticas de programação aplicadas no desenvolvimento, sem adentrar detalhes em algumas lógicas desenvolvidas no projeto, pois sairia do escopo deste trabalho, em razão do tamanho do projeto, que é composto por mais de 100 classes de *C#*.

Os exemplos de códigos neste capítulo foram implementados no jogo, porém eles estão sendo apresentados neste trabalho de forma simplificada e com os comentários em português, para facilitar o entendimento, não correspondendo necessariamente ao código final do projeto.

5.1 Soluções de problemas

Assim como o desenvolvimento de qualquer *software*, o desenvolvimento de um *game* consiste em: entender e compreender a aplicação que será implementada e o que deve ser feito; elaborar lógicas de programação para atingir os objetivos e solucionar problemas; e testar essas soluções para validar se elas realmente funcionam. Depois, esse processo se repete até a conclusão do desenvolvimento do sistema.

As soluções de problemas por meio da lógica, não envolve somente o conhecimento dele e da linguagem de programação que será usada. Envolve também o conhecimento de boas práticas de programação, o entendimento de como solucionar o problema, de forma mais eficiente, além de pesquisas e leituras das documentações e ferramentas que estão sendo utilizadas no desenvolvimento.

5.1.1 Dividir para conquistar

Esta prática é bastante difundida por programadores e consiste em reduzir um problema maior em subproblemas, que são mais simples de serem resolvidos, e a combinação dessas soluções é capaz de resolver o problema maior. Além disso, essa

prática permite simplificar os *scripts*, tornando-os mais simples de serem compreendidos e mais fácil de serem mantidos, de modo que eles possuiriam uma funcionalidade mais simples.

É válido ainda ressaltar, uma implementação utilizando essa prática, que corresponde à movimentação do jogador, como é demonstrado no segmento de código 3, em que demonstra a segmentação das lógicas de translação, rotação e animação do jogador.

Código 3 – Implementação da movimentação do jogador

```
      public class PlayerControllerScript : MonoBehaviour {
          // Componentes do jogador
          private Rigidbody rigidbody;
          // ...
          // Variável de rotação e movimentação
          private float horizontal, vertical, speed = 15f;
          private Joystick joystick;

          // Obtém-se a referência de alguns componentes.
          private void Awake () {
              // ...
          }

          private void Update () {
              // Obtém o valor do deslocamento vertical e horizontal do
joystick
              horizontal = joystick.Horizontal;
              vertical = joystick.Vertical;

          }

          private void FixedUpdate () {
              PlayerMovement ();
              PlayerRotation ();
              PlayerAnimation ();
          }

          // Este método é responsável por fazer o jogador se mover.
          private void PlayerMovement () {
              Vector3 movePosition = new Vector3(-vertical, 0.0f,
horizontal).normalized * speed * Time.deltaTime;
              rigidbody.MovePosition (transform.position + movePosition);
          }

          // Este método faz o avatar do jogador rotacionar de acordo com o
deslocamento do joystick virtual
          private void PlayerRotation () {

              // Apenas rotaciona o jogador se ele estiver se movendo
              if (vertical != 0f || horizontal != 0f) {
                  // Lógica da rotação ...
                  // ...
              }
          }

          // Este método é responsável por controlar os estados de animação do
avatar do jogador.
          private void PlayerAnimation () {
              // Lógica de animação
              // ...
          }
      }
```

Fonte: Concebido pelo autor

5.1.2 Manipulação de componentes

Frequentemente, é realizado o controle dos componentes de *GameObjetcts*, a partir de *scripts*, para garantir que determinada funcionalidade dos objetos possa ser alterada com a lógica do jogo. Como por exemplo, em um jogo de tiro, em que o jogador realiza disparos em inimigos, espera-se que os disparos causem danos aos inimigos atingidos, a partir da lógica implementada. Uma das formas de realizar isso, é buscando a referência de componentes, como *scripts* de saúde do inimigo, para ter acesso a métodos e campos públicos os quais possam afetar a saúde deles.

Na *Unity*, a forma de acessar outros componentes através de um *script*, tanto no próprio *GameObject* quanto em outros, é utilizando o método *GetComponent*. Esse método, retorna uma referência de um componente do tipo especificado, caso exista, do contrário, retorna o valor nulo (*null*).

5.1.2.1 Comunicação entre *scripts*

A comunicação entre *scripts* ocorre quando se busca a referência de um componente de *script*, permitindo ter acesso aos métodos e variáveis públicas. Sendo, muitas vezes, fundamental para a implementação da funcionalidade e coerência das mecânicas do jogo, principalmente quando há alguma relação lógica entre diferentes elementos dele.

O exemplo do segmento de código 4 corresponde a uma demonstração de um *script* utilizado para bonificar o jogador ao coletar o *GameObject* que possui este componente vinculado.

Código 4 – Comunicação entre *scripts* para recompensar o jogador

```
      // Este script é responsável por guardar informações e status do
jogador, como quantidade de moedas, experiência e nível de jogador.
      private PlayerManagerScript playerManager;

      // Valor, em moedas, que este objeto vale quando é coletado
      int moneyValue = 10;

      // Este método é invocado quando um objeto (other) entra em contato
com o Trigger (componente Collider) deste objeto
      void OnTriggerEnter (Collider other) {
          // Verifica se a tag do objeto é igual a "Player"
          // Em outras palavras, verifica se foi o jogador que entrou em
contato com o objeto
          if (other.tag == "Player") {

              // Pega a referência do componente PlayerManagerScript do
Player
              playerManager = other.GetComponent<PlayerManagerScript>();

              // Chama o método que adiciona dinheiro ao jogador
              playerManager.ChangeMoney(PlayerManagerScript.GetMoney() +
moneyValue);

              // Destrói este objeto, dando a sensação de que ele foi
coletado.
              Destroy (this.gameObject);
          }
      }
```

Fonte: Concebido pelo autor

Outro exemplo de *script* é utilizado para o carregamento de arquivos de "*save game*", que permitem o usuário retomar o jogo de onde parou, após fechar a aplicação, sem precisar recomeçar tudo do zero.

Código 5 – Comunicação entre *scripts* para o *save game*

```
      // ...
      // Este script é responsável por carregar o "save game"
      private LoadData loadScript;

      // Este método é executado no primeiro frame
      private void Start () {
          // Busca a referência de um GameObject com o nome "SaveManager"
          GameObject saveManager = GameObject.Find("SaveManager");

          // Verificar se existe se esse objeto existe e pega o componente
LoadData
          if (saveManager != null)
              loadScript = saveManager.GetComponent<LoadData>();

      }

      // Este método é invocado por um clique de um botão e chama o método
Load do script LoadData.
      public void TaskOnClick () {
          if (saveManager != null)
              loadScript.Load(); // Carrega os dados do arquivo do save
game
      }

      // ...
```

Fonte: Concebido pelo autor

Além disso, outra estratégia de realizar comunicação de *scripts* é através de eventos, em que é possível invocar vários métodos, em diversos *scripts*, de uma só vez, apenas com a ocorrência de um evento. Uma implementação do jogo utilizando esta abordagem foi a lógica de bloquear o movimento do jogador ao iniciar um diálogo, sendo assim, no momento em que a janela de diálogo aparece, o botão de analógico fica inativo e o jogador não é capaz de se mover. Os segmentos de código 6 e 7 apresentam esta lógica em mais detalhes.

Código 6 – Implementação de eventos

```
      /// <summary>
      /// Este Script armazena o evento de abrir o painel de janela de
diálogos.
      /// Outros Scripts podem atribuir alguns métodos a serem chamados
quando o evento ocorrer.
      /// Use "+=" para associar o método para este evento e "-=" para
desassociar o método deste evento.
      /// Cuidado: Quando um método é associado ao evento, é uma boa prática
ter que desassociar em determinadas condições,
      /// como OnDisable ou OnDestroy, isto é, quando o objeto não existir
mais ou for desabilitado.
      /// </summary>
      /// <example>
      /// private void OnEnable () {
      ///          OpenDialogueEvent += SomeMethod;
      ///          CloseDialogueEvent += OtherMethod;
      /// }
      /// private void OnDisable () {
      ///          OpenDialogueEvent -= SomeMethod;
      ///          CloseDialogueEvent -= OtherMethod;
      /// }
      /// </example>
      public class WindowEvent : MonoBehaviour {
          // Declaração dos eventos
          public delegate void OpenWindow ();
          public static event OpenWindow OpenWindowEvent;

          // ...

          // Ao abrir a janela de diálogo, invoca o evento
"OpenWindowEvent".
          private void OnEnable () {
              if (OpenWindowEvent != null)
                  OpenWindowEvent (); // Invoca o evento
          }

          // ...

      }
```

Fonte: Concebido pelo autor

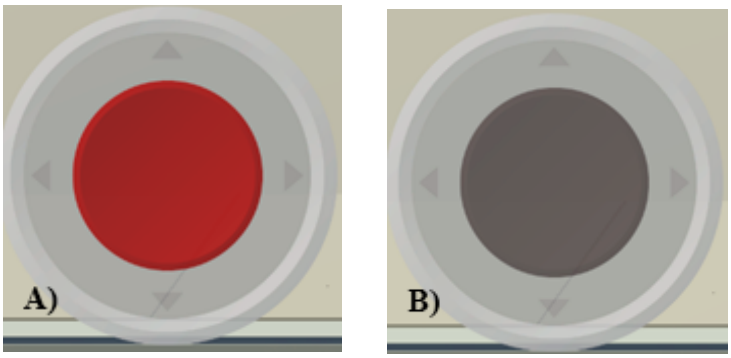
Código 7 – Chamada de métodos a partir de eventos

```
      /// <summary>
      /// Esta classe é responsável por gerenciar o joystick do jogo, como
quando ele deve ser habilitado ou desabilitado.
      /// </summary>
      public class JoystickManager : MonoBehaviour {

          private Joystick joystick;
          // ...

          private void Start () {
              WindowEvent.OpenWindowEvent += DisableJoystick;
              // ...
          }

          private void OnDisable () {
              WindowEvent.OpenWindowEvent -= DisableJoystick;
              // ...
          }


          public void DisableJoystick () {

              // Pega os componentes de imagem do analógico
              Image background = joystick.background.GetComponent<Image> ();
              Image handle = joystick.handle.GetComponent<Image> ();

              // Torna o analógico irresponsivo.
              background.raycastTarget = false;
              handle.raycastTarget = false;

              // Altera a cor do analógico (feedback visual)
              Color disabledColor = new Color (0.28f, 0.22f, 0.22f, 0.8f); //
Define uma nova cor (r, g, b, a)
              handle.color = disabledColor;

              // ...
          }

      }
```

Fonte: Concebido pelo autor

As figuras 48(a) e 48(b) demonstram o resultado desses *scripts*, apresentando o *feedback* visual do *joystick* no jogo, implementado no código 7.

Figura 48 – Bloqueio do *joystick*



Fonte: Concebido pelo autor

5.1.3 Sincronização de *scripts*

Durante a execução do jogo, múltiplos *scripts* estão sendo executados ao mesmo tempo, de forma paralela, porém algumas vezes, é necessário garantir que certa lógica seja executada antes da outra. Como a *Unity* não suporta programação em *Threads*, é possível sincronizar certos trechos de códigos utilizando *Coroutine*.

Um exemplo, implementado no *game*, ocorria quando se carregava o arquivo de salvamento do jogo, enquanto os dados estavam sendo lidos do arquivo, o cenário estava sendo carregado, porém era necessário garantir que os estados do jogo fossem atualizados, antes que o jogador pudesse começar a interagir. Para acrescentar um pequeno atraso em lógicas, que deveriam ser executadas somente depois da atualização do estado, foi utilizado *Coroutine*, suspendendo a execução de alguns trechos de código, por um pequeno período, garantindo que a lógica fosse executada com os dados atualizados.

Uma das lógicas em que essa estratégia foi aplicada, diz respeito aos objetos coletáveis, que podem ser recolhidos pelo jogador, como moedas. Quando se carregava o jogo, a partir de um "*save game*", era necessário garantir que objetos coletados pelo jogador não aparecessem novamente. Porém, esses objetos só aparecem depois que a cena é carregada, sendo necessário um atraso no *script* que desabilitava estes objetos, uma vez que ele precisa esperar que a cena seja carregada para poder desabilitar os objetos específicos.

5.2 Manutenibilidade de código

Um fator fundamental para agilizar o processo de desenvolvimento e manutenção de *software* é escrever códigos, os quais prezem por manutenibilidade, que consiste em torná-los mais fácil de serem modificados, mantidos, preservando a legibilidade, que consiste em escrever códigos, que sejam fáceis de serem compreendidos. Algumas práticas, as quais facilitam a manutenção dos códigos e do projeto serão descritas nesta seção.

5.2.1 Padrões de nomenclatura

A padronização de nomes de métodos e variáveis é de grande importância para facilitar a escrita de códigos futuros, uma vez que não será necessário pesquisar ou memorizar a maneira como os nomes de variáveis e métodos foram escritos. Além de estabelecer que se deve utilizar nomes expressivos, descrevendo a finalidade do método ou variável, deixando o código mais legível.

Para exemplificar, imagina-se que é estabelecido um padrão de nomenclatura em *camel case*, em que as primeiras letras de palavras compostas são maiúsculas. Se todo o código seguir este padrão, o desenvolvedor saberá como foi escrito os nomes das variáveis ou métodos, declarados por ele ou por outros desenvolvedores. Caso algum desenvolvedor não siga este padrão, poderá dificultar o desenvolvimento e comunicação entre *scripts*, uma vez que vai exigir a consulta e busca em outros *scripts*.

5.2.2 Padrões de comentários

No intuito de facilitar o entendimento dos *scripts* e evitar a necessidade de abrir e ler o código em detalhes, para entender a sua lógica. Foi definido que os *scripts* fossem escritos em inglês e contivessem comentários descrevendo sua funcionalidade, bem como

comentários mais detalhados para códigos mais complexos, como este apresentado no segmento de código 8.

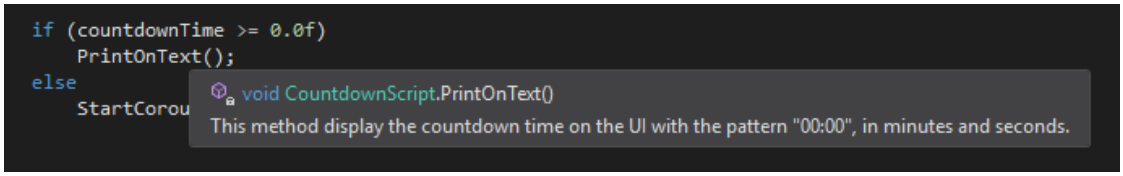
Código 8 – Padrões de comentários

```
        // ...


        /// <summary>
        /// This method displays the countdown time on the UI with the
pattern "00:00", in minutes and seconds.
        /// </summary>
        private void PrintOnText () {
            int minutes = (int) countdownTime / 60, seconds = (int)
countdownTime % 60;
            string textMinutes = (minutes / 10 > 0) ? minutes.ToString() :
"0" + minutes;
            string textSeconds = (seconds / 10 > 0) ? seconds.ToString() :
"0" + seconds;
            countdownText.text = textMinutes + ":" + textSeconds;
        }

        // ...
```

Fonte: Concebido pelo autor

Todos os comentários realizados nos códigos do jogo seguiram este padrão, que é reconhecido e apresentado pelo *IntelliSense* do *Visual Studio*, esta ferramenta é utilizada para auto completar códigos, além de fazer sugestões e apresentar descrições de métodos. Então, quando se escreve a chamada do método é apresentado a descrição dele. A figura 49 apresenta a descrição do método implementado no segmento de código 8.

Figura 49 – Descrição de método pelo *IntelliSense*



Fonte: Concebido pelo autor

5.2.3 Refatoramento e simplificação de código

Manter os *scripts* simples melhora bastante a sua legibilidade e manutenibilidade, sendo mais favorável ter vários *scripts* que realizam uma tarefa simples do que poucos, mas grandes, *scripts* que realizam múltiplas tarefas, pois dificulta a localização de falhas na lógica e entendimento do código, podendo ser necessário reescrever alguns trechos ou o próprio código inteiro.

Um exemplo de reescrita e simplificação de código no jogo foi realizado em um *script* implementado de forma inapropriada, contendo múltiplas lógicas e uma extensão de mais de 200 linhas de código, além de não possuir comentários e apresentar nomenclaturas de variáveis e métodos pouco expressivos, sendo necessário uma análise cuidadosa para compreender a lógica do código.

Um dos *scripts* refatorados foi a classe *SaveData*, que era responsável por salva o estado do jogo no arquivo, carregar o estado do jogo (*Load*) e formatação do arquivo de *save*, além de acessar diversas variáveis públicas de vários *scripts*, tanto para a obtenção quanto para alteração dos valores. Como pode ser percebido, este *script* estava realizando múltiplas tarefas desassociadas, podendo ser decomposto em *scripts* menores e mais simples.

Alguns dos refatoramentos que foram realizados consistia em manter o encapsulamento dos campos, que antes eram públicos e acessados por esta classe, isolando as alterações destes campos, dentro de suas respectivas classes, garantindo um melhor controle sobre suas alterações. Além disso, foi realizado a adição de comentários e decomposição deste *script* em 3 menores: um responsável pelo salvamento do jogo (*SaveData*), composto por cerca de 70 linhas de código; outro responsável pelo carregamento dos valores dos arquivos de *save* (*LoadData*), possuindo 50 linhas de código; e, por fim, mais um responsável pelos valores de armazenamento do arquivo (*SaveDataFile*), sendo esta classe composta por 30 linhas de código.

6 OTIMIZANDO O JOGO

A otimização de um jogo digital é uma questão fundamental no desenvolvimento do jogo, uma vez que ela define quais tecnologias vão suportar o jogo, além de ser um mecanismo importante para motivar a simplificação do projeto. De acordo com Gonzalez (2017), quando um *video game* é desenvolvido com otimização em mente, deve-se analisar o problema e avaliar o que deve ser feito na forma mais simples possível, evitando-se excessos de tarefas que aumentem o custo computacional da solução. Desse modo é possível reduzir riscos e evitar retrabalho, além de manter o jogo mais consistente durante o seu desenvolvimento.

Uma boa otimização pode ser a chave para o sucesso de um jogo, pois garante uma melhor fluidez e responsividade, além de ampliar a abrangência de *hardwares* capazes de rodar este jogo. É importante reiterar aquilo que foi apresentado por Turner e Schell (2016), que afirmava que os problemas de *performance* costumam ser únicos para cada jogo, o que implica dizer que nem todas as soluções que funcionaram para outros jogos vão funcionar no jogo que está sendo desenvolvido, podendo desperdiçar esforços da equipe e causar impactos negativos ao jogo.

Mesmo com os grandes avanços na tecnologia de *smartphones*, em termos de recursos e capacidade de processamento, com aparelhos como o recente *Galaxy* S10, da *Samsung*[18], possuindo um processador de oito núcleos (*octa core*), 8 *GB* de memória *RAM* e mais de 128 *GB* de armazenamento. Ainda é necessário realizar um processo de otimização especial para os dispositivos móveis, devido ao fato que muitos dos avanços tecnológicos estão presentes apenas em celulares de ponta (*High end*), enquanto a maioria dos aparelhos móveis ainda apresenta muitas restrições na capacidade de *Hardware*, como *CPU*s e *GPU*s de baixa capacidade de processamento, quantidades de memória e armazenamento muito limitadas.

Além disso, Cohade (2015) afirma que otimizações em jogos apresentam melhorias na eficiência energética da aplicação, o que implica em um menor consumo de bateria (em aparelhos móveis), menor aquecimento de *hardware* e um uso mais

[18] Disponível em: https://www.samsung.com/br/smartphones/galaxy-s10/specs/

prolongado da aplicação no aparelho, podendo garantir uma experiência mais agradável e a satisfação do jogador.

Neste capítulo, será apresentado e enfatizado algumas das estratégias de otimização aplicadas no desenvolvimento do “Recruta Social”.

6.1 Otimização de renderização

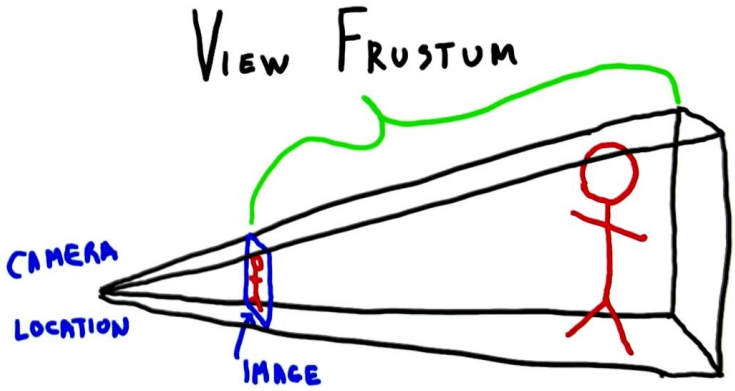
O aspecto visual (gráficos) do jogo pode ser um dos pontos que mais requer cuidados, ajustes e otimização para se obter uma melhor performance. Os elementos visuais do jogo precisam ser desenhados na tela, várias vezes, e um grande número desses elementos afetam o número de chamadas de funções que os desenha na tela (draw calls), tais funções podem ser encontradas em *API*s gráficas, como *OpenGL* e *DirectX*. Em suma, essas funções possuem instruções que devem ser executadas pela *GPU*, apresentando informações do que deve ser desenhado na tela.

Muitas vezes o processo de renderização pode ser bastante custoso, seja na parte da criação dos dados que serão utilizados para a síntese de imagem, a partir da *CPU*, seja no próprio processo de síntese de imagem, realizado pela *GPU*. Desse modo é importante adotar estratégias que simplifiquem esse processo, com o intuito de torná-lo mais eficiente. Algumas destas estratégias serão discutidas nesta seção.

6.1.1 Culling Frustum

Como discutido na seção 2.2.5.1, o *Camera Frustum* consiste em um volume 3D que corresponde ao campo de visão da câmera. Esse volume tem formato de pirâmide, sem o cume, para câmeras com projeção em perspectiva, como é demonstrado na figura 50.

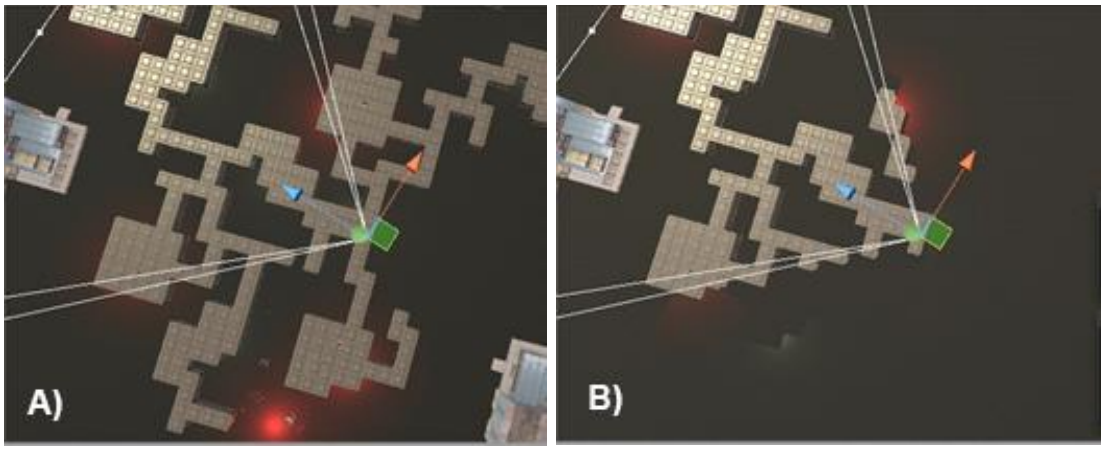
Figura 50 – Representação do *Camera Frustum*



Fonte: Canal do *YouTube* "Udacity"[19]

O *Frustum Culling* determina que objetos fora do volume do frustum da câmera não devem ser renderizados, como é demonstrado na figura 51(b). Desse modo, é possível diminuir o número de chamadas de desenho (*Draw Calls*) e melhorar o desempenho do jogo. Em princípio, a *Unity* utiliza esse mecanismo em objetos de câmeras virtuais.

Figura 51 – Demonstração do *Frustum Culling*



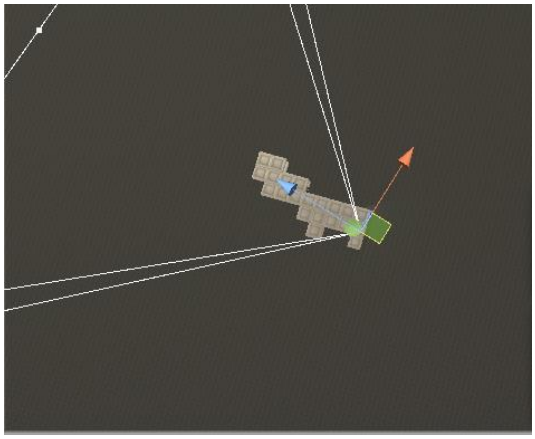
Fonte: Documentação da *Unity*[20]

Essa técnica pode ser intensificada com a utilização de outro mecanismo, denominado *Occlusion Culling*, que consiste em não renderizar objetos os quais estão

[19] Disponível em: https://www.youtube.com/watch?v=GqEP79loyQE
[20] Disponível em: https://docs.unity3d.com/Manual/OcclusionCulling.html

sendo obstruídos da visão da câmera. Por exemplo, supõem-se que há um objeto está sendo obstruído por uma parede, ou seja, a parede é opaca e está posicionada entre ele e a câmera do jogo. Dessa forma, do ponto de vista da câmera, não será possível enxergar esse objeto, então, com *Occlusion Culling*, o objeto em questão deixaria de ser renderizado, mesmo estando dentro do *Frustum* da câmera virtual. Uma demonstração da aplicação de *Occlusion Culling* pode ser vista na figura 52.

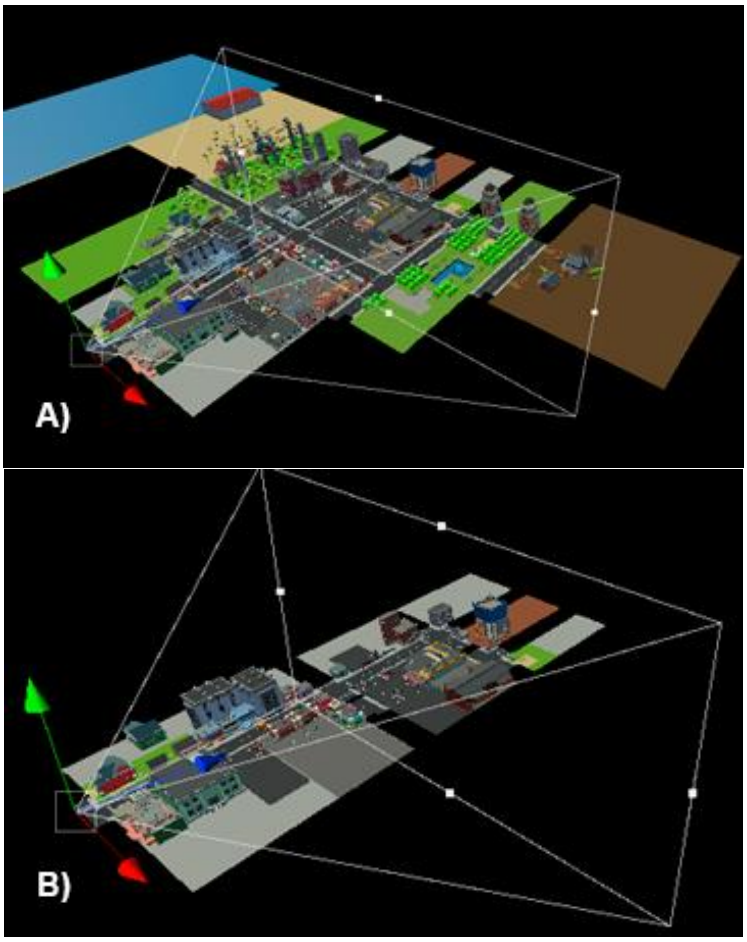
Figura 52 – Demonstração do *Occlusion Culling*



Fonte: Documentação da *Unity*

A figura 53(a) apresenta a porção de elementos do cenário da cidade que estão sendo renderizados, considerando apenas os objetos que estão dentro do *Frustum* da câmera. Enquanto que a figura 53(b) demonstra o resultado utilizando o mecanismo de *Occlusion Culling*, demonstrando que apenas os objetos que são visíveis no ponto de vista da câmera estão sendo desenhados.

Figura 53 – Comparativo de *Culling* na aplicação



Fonte: Concebido pelo autor

6.1.2 Agrupamento de objetos

De acordo com Gonzalez (2017), o agrupamento de objetos (*batching*) consiste em um mecanismo da *Unity* de simplificar o processo de renderização a partir do aglomerado de objetos que compartilham características em comum, como o seu material e textura. Com isso, é possível desenhar esses objetos em lotes, permitindo reduzir o número de *Draw Calls*. Na *Unity* há dois tipos dessa técnica: O agrupamento estático (*Static Batching*) e o agrupamento dinâmico (*Dynamic Batching*).

Os objetos estáticos consistem em elementos no cenário que não sofrem operações lineares, como rotação, translação e escalonamento. Em outras palavras, esses objetos são fixos e não devem ser alterados. Dessa forma, o *Static Batching* é realizado em objetos

estáticos que compartilham o mesmo material. De modo semelhante, o *Dynamic Batching* é realizado em objetos dinâmicos (não-estáticos) que compartilham o mesmo material.

Assim é possível agrupar as malhas de objetos estáticos, como demonstrado na figura 54, e simplificar a renderização de objetos dinâmicos. É importante ressaltar que agrupamentos estáticos são mais eficientes do que agrupamentos dinâmicos, porém ocupam mais memória, para armazenar as malhas combinadas.

Figura 54 – Demonstração de agrupamento de malhas

Fonte: Concebido pelo autor

6.2 Otimização do uso de memória

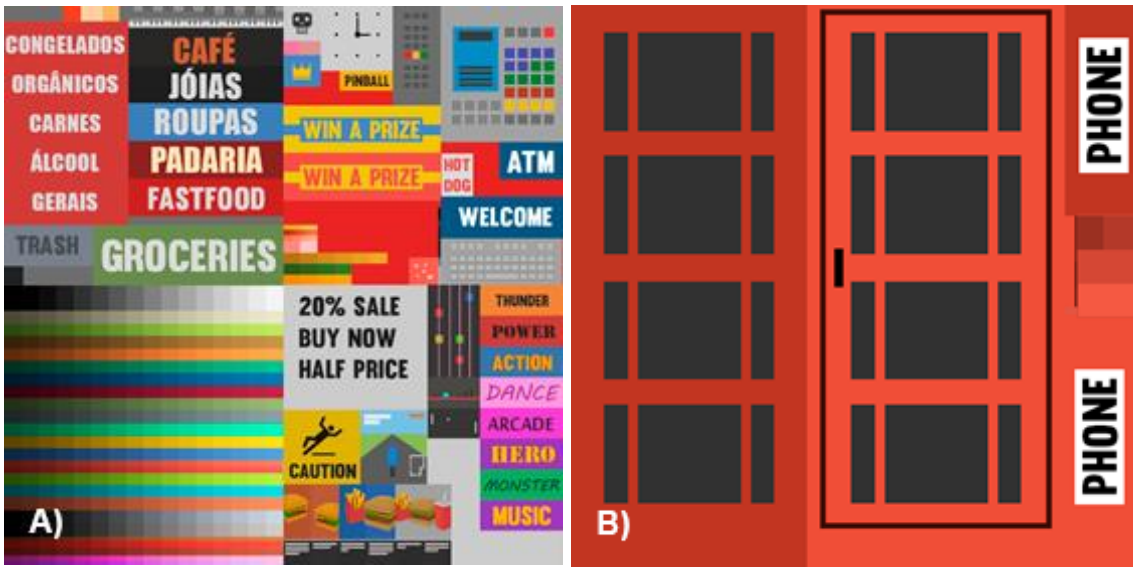
As otimizações em termos de uso de memória costumam envolver (de forma geral) os *assets* (recursos) utilizados para o desenvolvimento do jogo, como modelos 3D, texturas, textos, áudio e imagens. Dito isso, alguns dos ajustes realizados para obter um uso mais eficiente de memória foram: reuso e redução de texturas; otimizações de modelos; redução de usos de algoritmos que utilizem mais memória.

6.2.1 Reuso de texturas

É comum encontrar objetos que utilizam diferentes mapas de texturas (figura 55(b)), sejam eles de baixa ou alta resolução. Utilizar uma variedade de texturas de alta definição na cena do jogo pode exigir um maior consumo de memória e causar impactos significativos na *performance*, podendo impor cuidados extras.

Embora ainda seja possível manter a utilização de várias texturas, há alternativas para evitar ou mitigar problemas que podem ser causados por essa abordagem. Uma dessas alternativas, apresentada por Gonzalez (2017), consiste em utilizar mapas de texturas virtuais (ou megatexturas), que consistem em imagens que contém vários mapas de texturas combinados (figura 55(a)). Podendo ser aplicado em muitos objetos ao mesmo tempo, o que facilita o agrupamento, pois eles compartilham o mesmo material. Essa técnica é muito utilizada em jogos de grandes orçamentos (AAA), além de facilitar a organização e reduzir o número de texturas.

Figura 55 – Megatextura e mapa de textura simples



Fonte: Concebido pelo autor

6.2.2 Redução de texturas

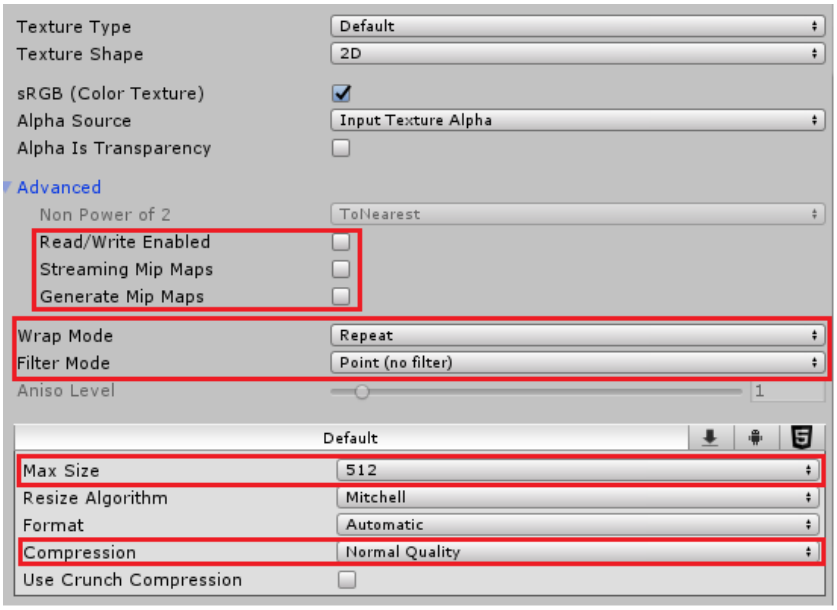
Segundo Dundore e Harkness (2016), algumas otimizações que podem ser realizadas em texturas utilizadas pela *Unity* consistem em:

- Desabilitar a propriedade de Leitura/Escrita (Read/Write), evitando guarda duas cópias dessa textura na memória;
- Desabilitar *mipmaps*;
- Utilizar algoritmos de compressão nas texturas;

- Limitar o tamanho das texturas. Para o “Recruta Social”, o tamanho máximo estabelecido foi 512x512.

A figura 56 apresenta, no editor da *Unity*, o resultado das modificações descritas para as propriedades.

Figura 56 – Propriedades de texturas (editor da *Unity*)



Fonte: Concebido pelo autor

6.2.3 Otimizações de modelos

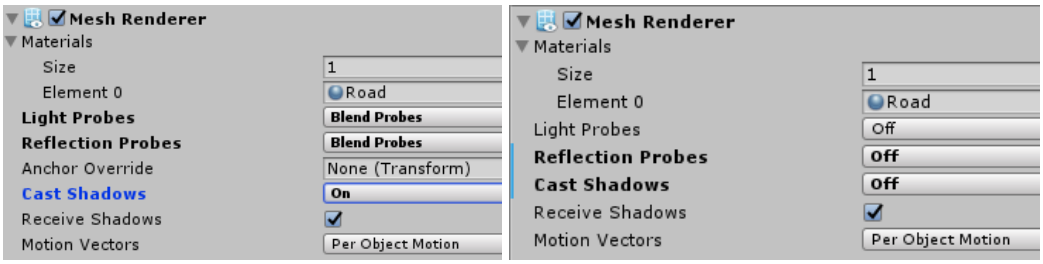
Para a otimização de modelos, Dundore e Harkness (2016), apontam as seguintes estratégias:

- Desabilitar a propriedade “Read/Write”;
- Evitar a utilização de *rig* (esqueleto de animação) em modelos que não possuem animação;
- Compartilhar o avatar de animação entre modelos que possuem *rigs* compartilhados;
- Aplicar compressão de modelos;
- Habilitar a propriedade de otimizar a malha;

Outra otimização realizada nos modelos está relacionada ao componente *Mesh Renderer*, que é responsável por renderizar o objeto na cena do jogo, de acordo com a sua posição, rotação e escala, que são definidas no componente *Transform*. No componente *Mesh Renderer* há algumas propriedades que podem ser desabilitadas para melhorar o desempenho, como desabilitar projeção de sombras e reflexão (ver figura 57).

Essas propriedades costumam vir habilitadas (por padrão) nos objetos que são renderizados, podendo ser necessário desabilitar essas propriedades em alguns elementos que possuem vários modelos no cenário, como grama, ou que não apresentam muita diferença em utilizar essas propriedades habilitadas, como por exemplo, o chão do cenário projetar sombra.

Figura 57 – Alterações no *Mesh Renderer*



Fonte: Concebido pelo autor

6.3 Otimização da física

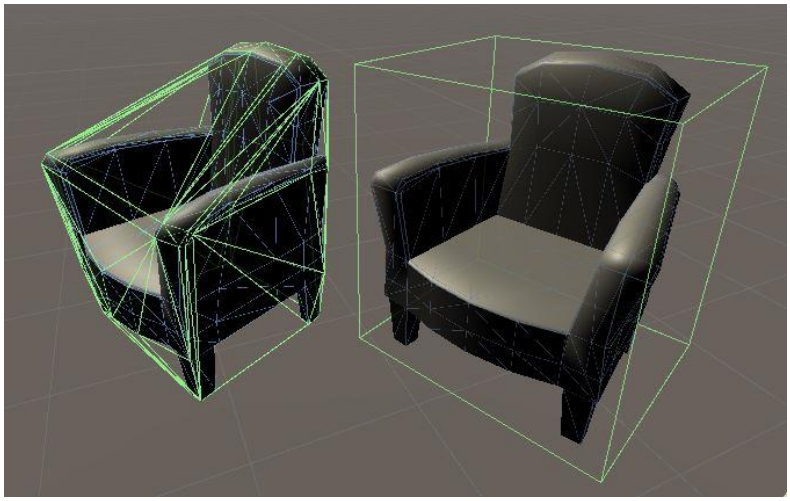
Os cálculos de física são amplamente utilizados em diversos tipos de jogos, sejam eles feitos na *Unity* ou em outro motor de jogo. Nesse aspecto, pode haver simplificações e reduções de custos dos cálculos realizados para simular a física.

Algumas das medidas tomadas no desenvolvimento do “Recruta Social” foi utilizar colisores (*colliders*) primitivos, como colisor de caixa (*box collider*), ao invés de colisores mais complexos, como o de malha (*mesh colliders*), uma vez que esses possuem um maior número de polígonos, podendo exigir cálculos mais complexos.

Para atestar a ineficiência do *mesh collider*, Matt (2016) realizou um experimento que consistia em um comparativo entre o *box collider* e o *mesh collider*, utilizando uma distribuição uniforme de 1600 poltronas (*grid* de 40x40) da figura 58. O resultado obtido

apontou que os *mesh colliders* podem chegar a ser 20 vezes mais lentos em relação ao *box colliders*.

Figura 58 – *Mesh Collider* e *Box Collider*



Fonte: Blog "RogueCode"[21]

Além disso, outra otimização realizada na física do jogo foi o uso reduzido de *Rigidbody*, pois grandes quantidades de objetos com esse componente podem causar impactos significativos à *performance* do jogo, de modo que cada um desses objetos será considerado nos cálculos da simulação de física, como aplicação de força, aumentando o *overhead* destes cálculos.

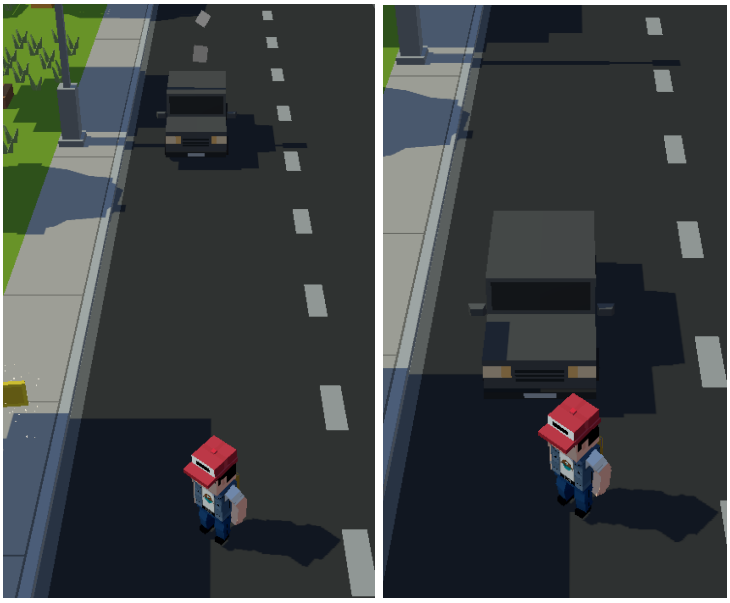
Para reforçar, o *Rigidbody* é um componente da *Unity* que deve ser atribuído aos objetos do jogo (*GameObject*) que devem ser afetados pelo sistema de física. Sendo fundamental para a simulação de movimentos realistas baseados em física, além de permitir a detecção de colisão com outros objetos.

Em alguns casos, o uso desse componente é feito apenas para detecção de colisão entre objetos, desconsiderando-se o comportamento realista da simulação física, com o intuito de reduzir o custo dos cálculos de física e melhorar o desempenho do jogo. Para isso, é preciso habilitar uma propriedade do *Rigidbody*, denominada "*isKinematic*", que indica se este objeto deve, ou não, ser afetado pelo sistema de física da *Unity*.

[21] Disponível em: http://blog.roguecode.co.za/Unity3D-Mesh-Collider-vs-Box-Collider

A figura 59 demonstra um exemplo aplicado nos veículos do jogo, em que esses objetos possuem o componente *Rigidbody* com a propriedade "*isKinematic*" habilitada, fazendo com que eles não sejam afetados pela física. Dessa forma, esse componente está sendo utilizado apenas para a detecção de colisão, alterando o comportamento do objeto ao entrar em contato com outro. No caso em questão, o carro freia ao se deparar com o jogador.

Figura 59 – Comportamento dos veículos



Fonte: Concebido pelo autor

6.4 Otimização de código

No desenvolvimento de jogos digitais, é muito recorrente acontecer a implementação de *scripts* ineficientes (em termos de desempenho), seja devido a lógica criada no código ou por fazer uso de métodos e *API*s que são bastante custosas. Tendo isso em mente, este tópico vai abordar algumas estratégias utilizadas para melhorar o desempenho de códigos no jogo.

6.4.1 Limpeza de Código

A simplificação de código é uma das melhores formas de garantir que ele seja eficiente, alguma de manter o *script* simples consistem em: manter no método *Update* apenas as lógicas que precisam ser executadas e atualizadas constantemente; utilizar *Coroutines*, como uma alternativa ao *Update*, para as lógicas que devem ser executadas em tempos regulares, permitindo reduzir a quantidade de instruções que são executadas continuamente.

6.4.2 Uso de *API*s custosas

De acordo com Dundore e Harkness (2017), a *Unity* pode fornecer várias soluções em suas *API*s que são consideradas ineficientes em termos de desempenho, seja por tempo de processamento, seja por consumo de memória. Tendo isso em mente, é importante tomar alguns cuidados com o uso de alguns métodos de *API*s que, mesmo podendo ser convenientes, ocasionam aumentos significativos no tempo de processamento, comprometendo o funcionamento do jogo. Isso não implica dizer que esses métodos não devem ser utilizados, e sim que o seu uso deles deve ser moderado e consciente.

6.4.3 Métodos que retornam um vetor

Dundore e Harkness (2017) alertam para o uso de métodos que retornam um vetor (*array*), como o *Input.touches*, pois esses métodos geralmente geram uma cópia do *array* a ser retornado por questões de segurança, fazendo alocações temporárias na memória *Heap* a cada chamada desses métodos.

Como descrito na seção 2.4.1.4.3, a memória *Heap* é uma região da memória que contém dados que são alocados dinamicamente nos códigos. Além disso o seu tamanho pode aumentar na medida em que mais espaço é demandado pela aplicação, durante o seu tempo de execução, tendo como o limite de espaço a capacidade física de memória. Sendo

assim, a ocupação pela memória *Heap* apenas cresce, a medida que é demandado, e nunca decai depois da expansão do espaço alocado, mesmo com a liberação com o *Garbage Collector*.

Dito isso, para reduzir o número de alocações dinâmicas na memória e, consequentemente, o processamento do *Garbage Collection* e ocupação na memória principal, é preciso reduzir o número de chamadas desses métodos. Uma abordagem de minimizar o número de chamadas é: guardar a referência dos valores retornados depois de uma invocação desses métodos, para ser utilizada posteriormente; usar uma variante desses métodos que não realize alocação dinâmica, como por exemplo, utilizar *Input.GetTouches* ao invés de *Input.touches*.

Para reforçar o entendimento dessa estratégia, será apresentado três variações de um segmento de código que realizam a mesma tarefa. O segmento de código 9 corresponde a primeira alternativa, sendo a mais ineficiente das três, uma vez que realiza várias alocações temporárias da lista de toques na tela (*Touches[]*), a cada vez que a instrução *Input.touches* é executada, o que ocorre tanto na verificação do tamanho do vetor, no laço de repetição (*for*), quanto no acesso por índice de um elemento do *array*.

Código 9 – Trecho de Código com várias alocações temporárias

```
      // Input.touches sempre retorna uma cópia de um array dos toque do
usuário.
      for(int i = 0; i < Input.touches.Lenght; i++){// Alocação dinâmica de
Touches[]
          Touch touch = Input.touches[i]; // Outra alocação dinâmica de
Touches[]
      }
```

Fonte: Concebido pelo autor

O segmento de código 10 apresenta a segunda alternativa, que faz um uso mais eficiente de memória em relação à primeira, pois é realizado apenas uma alocação dinâmica da lista de toques do usuário, que é reutilizada, posteriormente, dentro do laço de repetição.

Código 10 – Trecho de Código com uma alocação temporária

```
      // Esta alternativa só utiliza uma instância de Input.touches
      Touch[] touches = Input.touches; // Guarda a referência do array
alocado
      for (int i = 0; i < touches.Lenght; i++){
          Touch touch = touches[i]; // Reutiliza o array várias vezes
          // ...
      }
```

Fonte: Concebido pelo autor

Por fim, o segmento de código 11 demonstra a terceira alternativa, que não realiza nenhuma alocação dinâmica na memória. Pois está sendo utilizado uma variante da *API* que não realiza alocação temporária, isto é, não gera uma cópia do *array* que será retornado na chamada da função. O método *touchCount* retorna apenas um inteiro, indicando a quantidade de toques na tela, enquanto que o método *GetTouch* retorna um objeto do tipo Touch, especificado pelo índice passado como argumento.

Código 11 – Trecho de Código sem nenhuma alocação temporária

```
      // Esta alternativa não realiza nenhuma alocação dinâmica do vetor de
Touches[]
      int touchCount = Input.touchCount; // Retorna o número de toque na tela
      for (int i = 0; i<touchCount; i++) {
          Touch touch = Input.GetTouch(i); //Retorna uma referência do toque i.
          // ...
      }
```

Fonte: Concebido pelo autor.

Além disso, Dundore e Harkness (2016) afirmam que se deve evitar o uso da instrução *foreach*, devido ao fato que ela sempre aloca uma variável temporária (*Enumerator*), ocupando um pequeno espaço na memória *Heap*, para poder percorrer uma lista de objetos. Além disso, os laços de repetição realizados com *foreach* costumar ser (aproximadamente) duas vezes mais lentos do que *loops* correspondentes com o *for* tradicional. Esse atraso ainda pode ser mais significativo em laços de repetições aninhados.

Código 12 – Variante de laço de repetição com *for* e *foreach*

```
// Pega a referência de uma lista grande
var list = GetSomeLargeList();

// Laço de reetição com foreach
foreach (int number in list)  {
    // Realiza alguma lógica
}


// Laço de repetição com for tradicional (Preferível)
for (int i=0; i<list.Count; i++) {
    var number = list[i];
    // Realiza alguma lógica
}
```

Fonte: Concebido pelo autor

6.4.4 Minimizar chamadas de métodos custosos

Guardar a referência de componentes e objetos é uma das práticas mais utilizadas para otimizar código, ela deve ser emprega sempre quando for possível. No momento em que se realiza a busca por uma referência de um objeto, usando o método *Find* da classe *GameObject*, ou de um componente, usando o método GetComponent, armazene-a em uma variável, para evitar o custo de refazer as buscas posteriormente.

Isto pode garantir uma melhora significativa no desempenho dos *scripts*, uma vez que as buscas realizadas por métodos, como o *Find*, são bastante custosas, especialmente em cenários com uma grande quantidade de objetos (dezenas ou centenas de milhares), pois a busca realizada é linear e leva em consideração todos os objetos da cena, realizando comparações de textos (*string*) entre o argumento da função e o nome de cada objeto. Portanto não é desejável realizar buscas de objetos e componentes mais vezes do que o necessário, podendo ser minimizado o custo ao guardar as referências, retornadas pelas buscas, em variáveis.

A partir disso, no “Recruta Social”, as buscas por objetos e componentes são comumente realizadas apenas uma vez, guardando-se a referência do objeto retornado

pela busca, que usualmente é realizada nos métodos Start ou Awake, os quais são executados no início de cada cena do jogo.

De acordo com Dundore e Harkness (2017) realizar a procura de objetos por seu rótulo (*tag*) é mais eficiente do que realizar a busca pelo seu nome. Dessa forma, é possível ter uma redução no tempo de busca pelo o objeto, utilizando métodos como *FindGameObjectWithTag*, que realizam buscas de objetos pela sua *tag*, no lugar do *Find*, que realiza a busca do objeto pelo seu nome.

As buscas por rótulos são mais eficientes devido à redução do campo de procura, uma vez que não é necessário analisar cada objeto na cena, comparando os seus nomes, pois essa busca só retorna o primeiro objeto indexado pela *tag* passada como argumento. Por outro lado, se houver mais de um objeto com o mesmo rótulo, a busca pode acabar retornando resultados distintos, embora possa ser realizado a busca de todos os objetos com determinado rótulo.

O segmento de código 13 apresenta uma implementação no jogo, que consiste em pegar a referência de todas as moedas, as quais possuem o rótulo "*Coin*", e manter 20% delas habilitadas, enquanto o restante fica inativo e, consequentemente, não aparecerão no cenário do jogo.

Código 13 – Implementação de lógica que habilita 20% das moedas

```
        // Este método é chamado na inicialização
        void Start()
        {
            // ...

            // Pega a referência de todos os objetos com a tag "Coins"
(moedas)
            GameObject[] coins = GameObject.FindGameObjectsWithTag("Coin");
            int size = coins.Length; // quantidade de moedas
            float dropout = 0;

            // Mantêm 20% das moedas habilitadas, escondendo restante.
            for (int i = 0; i < size; i++) {
                // Retorna um valor aleatório entre 0 e 100.
                dropout = Random.Range(0, 100);

                // Se o valor for menos do que 20 mantenha o objeto da moeda
ativado
                if (dropout < 20f)
                    coins[i].SetActive(true);
                else // Caso contrário, desabilite-o
                    coins[i].SetActive(false);
            }
            // ...

        }
```

Fonte: Concebido pelo autor

6.4.5 Redução de escala

Algumas vezes o impacto do desempenho pode ocorrer devido a quantidade de objetos que executam um determinado *script* ao mesmo tempo. Então mesmo que esse código esteja bem otimizado, pode ser necessário implementar alguma lógica extra que reduza o impacto causado pela escala desse *script*.

No exemplo do jogo, foi implementado um *script* simples que realiza a movimentação e rotação de moedas pelo cenário. Embora a sua lógica fosse básica, a quantidade de objetos (cerca de 300) que estava executando-a ao mesmo tempo estava causando o impacto na *performance*, sendo necessário adicionar uma lógica extra para realizar a animação das moedas apenas quando elas fossem visíveis na câmera do jogo. O segmento de código 14 apresenta a implementação dessa lógica.

Código 14 – Implementação de animações em moedas no cenário

```
      /// <summary>
      /// Esse script é utilizado para realizer as animações das moedas no
jogo;
      /// </summary>
      public class CoinAnimation : MonoBehaviour {
      //...                                                                ...

// Este método é chamado quando o objeto se torna visível pela câmera do jogo.
          private void OnBecameVisible() {
              rotates = true; // Habilita a animação.
          }

// Este método é chamado quando o objeto não é mais visível pela câmera.
          private void OnBecameInvisible() {
              rotates = false; // Desabilita a animação.
          }

// Este método é chamado a cada frame
          void Update () {

               // Verifica se a animação está habilitada.
              if (rotates) {

                  // Faz a moeda rotacionar
                  newRotation = transform.localEulerAngles;
                  newRotation.y += 1f;
                  transform.localEulerAngles = newRotation;

                  // Faz a moeda subir/descer.
                  currentPosition = transform.position;
                  if (ascending) {
                      if    (currentPosition.y    <    originalPositon.y    +
maxRelativeYPosition)
                          currentPosition.y += movementRate;
                      else
                          ascending = false;

                  } else {
                      if    (currentPosition.y    >    originalPositon.y    -
minRelativeYPosition)
                          currentPosition.y -= movementRate;
                      else
                          ascending = true;
                  }

                  transform.position = currentPosition;

              }

          }
      }
```

Fonte: Concebido pelo autor

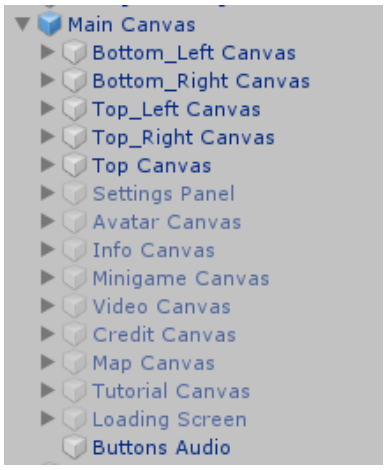
6.5 Otimização de performance na UI

Segundo Dundore e Harkness (2017), a interface de usuário em aplicações da *Unity* é um fator recorrente em causar impactos significativos no desempenho se não for implementado de forma eficiente. Por exemplo, se ocorrer alterações em um elemento da *UI* pode fazer com que toda a interface seja redesenhada. Esse fenômeno ocorre quando se utiliza apenas um *Canvas* na interface inteira e um de seus elementos é alterado. O que faz com que esse *Canvas* seja marcado como "sujo", ocasionando na geração das malhas e reagrupamento (*rebatching*) dos objetos dele.

Em outras palavras, basta que apenas um único elemento da interface seja alterado, para fazer com que os demais, que pertençam ao mesmo *Canvas*, sejam redesenhados, causando um grande impacto no desempenho. Pois é bastante custoso gerar os materiais e as malhas, além do custo adicional de redesenhá-las na tela. Esse processo é realizado para cada elemento ativo dentro desse *Canvas*, mesmo que ele seja transparente (*alpha* igual à zero).

Para minimizar esse efeito negativo, Dundore e Harkness (2017) apresentou uma estratégia que consiste em subdividir um *Canvas* em vários *sub-canvases*, de forma a agrupar elementos da interface que possuam alguma relação lógica, como, por exemplo, agrupar os elementos estáticos e agrupar elementos dinâmicos em *Canvas* separados. Além disso, é possível manter uma hierarquia de múltiplos *Canvases* aninhados, sendo um deles o objeto principal da hierarquia, o qual contém vários filhos na hierarquia, podendo ser elementos de UI (imagens, textos ou paineis) ou outros *Canvases* (*sub-canvases*). A figura 60 apresenta a hierarquia dos elementos da interface gráfica do "Recruta Social".

Figura 60 – Hierarquia da interface do “Recruta Social”



Fonte: Concebido pelo autor

Dessa forma, cada *Canvas* que é marcado como “sujo” só afetará os seus elementos “filhos”, enquanto que os elementos dos demais *Canvas* permanecerão intactos. Dessa forma, as alterações realizadas em cada *Canvas* são isoladas dos demais, mesmo que haja uma relação hierárquica entre ele, isto é, as alterações nos *sub-canvases* são isoladas do “pai”, e vice-versa. Então, tendo isso em mente, a interface do jogo (figura 61) foi desenvolvida utilizando vários *Canvases*, isolando, dessa forma, as alterações entre ele.

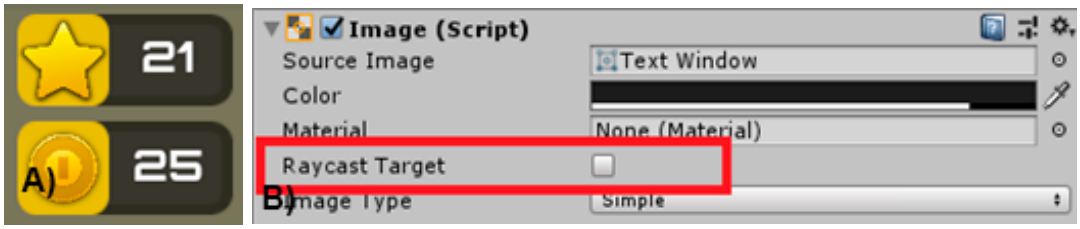
Figura 61 – Divisão dos *Canvases* na tela do jogo

Fonte: Concebido pelo autor

Por fim, outra melhoria que foi realizada na interface do jogo consiste em reduzir o número de elementos da *UI* que são *Raycast Target*, significando que esses objetos da interface são interativos, como botões, *sliders* ou uma caixa de texto. Como resultado é obtido uma redução no tempo de checagem realizado pelo *Graphic Raycaster*, cujo o funcionamento havia sido descrito na seção 2.3.7.2.

A propriedade "*Raycast Target*" vem habilitada por padrão quando se cria um elemento gráfico da interface, dessa forma pode ser necessário desabilitá-la, no caso de ser um elemento que não deva interagir com a entrada do usuário (como clique do *mouse* ou toque na tela). Um exemplo realizado no "Recruta Social" foi desabilitar a propriedade "*Raycast Target*" (figura 62(b)) em elementos que apenas apresentavam dados, como o *status*, quantidade de dinheiro e pontuação do jogador (figura 62(a)).

Figura 62 – Elementos não-interativos da *UI*



Fonte: Concebido pelo autor

6.6 Otimização de iluminação

A iluminação e sombreamento são um dos aspectos que podem ser mais complexos e pesados em um jogo eletrônico, principalmente para dispositivos móveis, que podem ter problemas com iluminação dinâmica. Devido ao fato que iluminação e projeção de sombras dinâmicas podem exigir que cálculos complexos sejam realizados continuamente, dependendo do nível de realismo desejado.

Dito isso, muitas vezes o ideal é simplificar este aspecto do jogo: reduzindo o número de fontes de luz, para alcançar o estilo desejado; ajustar qualidade de sombras (ou removê-las); utilizar uma combinação entre iluminação dinâmica e iluminação estática.

A iluminação estática consiste em gerar *light maps* para os objetos estáticos da cena, podendo trazer benefícios visuais ao jogo sem custo de processamento adicional, pois não exige o cálculo iluminação nesses objetos em tempo real. Porém esse mecanismo acarreta um maior uso de memória para armazena os *light maps*. A figura 63(a) representa o cenário da casa da tia apenas com iluminação dinâmica, enquanto que a figura 63(b) apresenta o mesmo cenário com iluminação mista (dinâmica e estática).

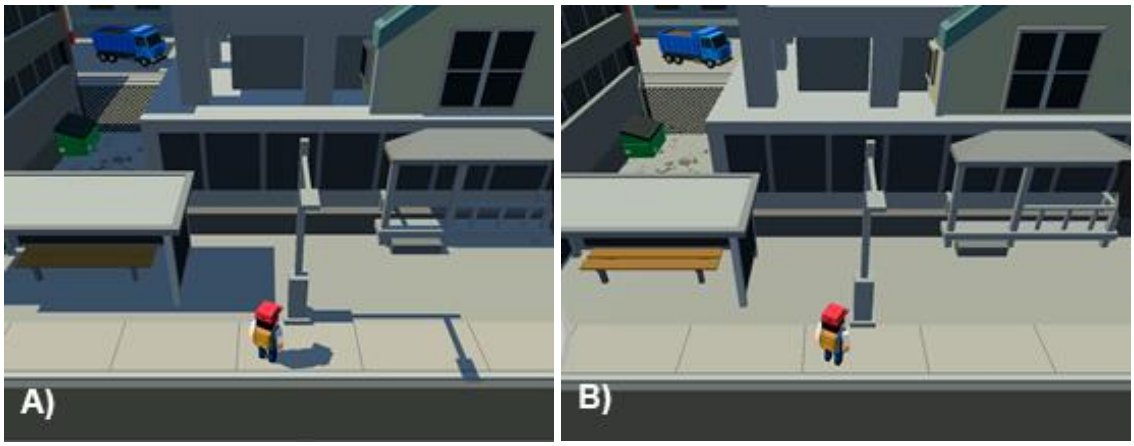
Figura 63 – Comparativo de iluminação dinâmica e mista

Fonte: Concebido pelo autor

Percebe-se, com a comparação entre as figuras 63(a) e 63(b), uma melhoria visual obtida com a *baked light* (iluminação estática). Além disso, houve também um ganho de desempenho, demonstrado pelo contador de *FPS* do editor da *Unity*, indo de 338 para 376 quadros por segundo. Sendo esse resultado não correspondente ao desempenho obtido em um dispositivo móvel.

Além disso, houve a adição de uma opção gráfica de baixa qualidade, permitindo ao usuário a possibilidade de escolher entre 2 opções gráficas, podendo garantir que ele possa ter uma melhor experiência de jogo com o seu aparelho, uma vez que ele será capaz de optar por melhores gráficos ou melhor *frame rate*. A diferença de qualidade gráfica pode ser observada nas figuras 64(a) (qualidade alta) e 64(b) (qualidade baixa).

Figura 64 – Comparativo entre as qualidade gráficas



Fonte: Concebido pelo autor

7 DESAFIOS ENCONTRADOS

Assim como diversos projetos de *software*, o projeto “Recruta Social” apresentou diversos desafios, além do seu próprio desenvolvimento. Alguns dessas dificuldades serão descritas brevemente neste capítulo.

7.1 Incerteza do cliente

Muitas vezes o TCE-PB não possuía uma visão bem definida sobre o que deveria ser o jogo, como seria os aspectos de *gameplay*, qual seria a forma de transmitir os objetivos instrucionais, quais elementos narrativos e problemas sociais deveriam ser contemplados no projeto. Tal contexto compromete todo o processo de desenvolvimento, uma vez que todas as questões levantadas são fundamentais para o andamento do projeto.

Como descrito no capítulo 4, muitos requisitos, aspectos de *gameplay* e narrativa foram concebidos e estabelecidos pela a equipe de desenvolvimento do projeto, o que exigia mais esforço e tempo por parte desta equipe. Posteriormente, era apresentado ao TCE-PB as propostas, e algumas vezes, já implementadas no jogo, porém houve casos que foi necessário refazer ou renunciar algumas dessas propostas para atender as expectativas do cliente, embora não ficasse muito claro a perspectiva que dele em relação ao projeto.

O desenvolvimento começou a ter avanços, na medida em que novas propostas, feitas pela equipe responsável pelo projeto, se alinhavam, de forma aproximada, com os interesses do Tribunal de Contas do Estado. Ao mesmo tempo que isso exigiu várias reuniões e sessões de *brainstorm*, foi possível elaborar uma proposta de jogo que pudesse atender as expectativas do cliente.

7.2 Controle de versionamento

Devido ao fato de o projeto ter sido realizado em equipe, foi necessário utilizar alguma ferramenta de gerenciamento de projeto e controle de versionamento,

possibilitando que cada membro pudesse trabalhar em uma versão do projeto, criando cópias e implementando as suas alterações, que posteriormente seriam integradas ao projeto principal.

Porém, esse contexto representou um grande desafio, uma vez que era muito recorrente acontecer conflitos entre as alterações. Estes conflitos ocorrem quando dois ou mais membros, trabalhando em suas respectivas cópias realizavam alterações em regiões comuns, por exemplo, se vários programadores alterassem os mesmos trechos de código de um determinado *script*, provocaria conflitos de projeto. Quando se realizava a combinação das alterações (*merge*), a ferramenta de versionamento (*git*) fundia os arquivos modificados para incluir as alterações feitas.

O maior desafio quando se realizava esta operação era lidar com conflitos dos próprios arquivos da *Unity*, principalmente os arquivos de cena (extensão ".unity"), não só devido a sua notação (YAML) como também o seu extenso tamanho, podendo passar de centenas de milhares de linhas, além de não ter muito controle sobre as alterações destes arquivos pela fusão das alterações, criando a possibilidade de invalidar o arquivo para a *Unity*.

7.3 Comunicação e engajamento da equipe

Alguns membros da equipe, que exerciam tarefas essenciais, tinham grandes dificuldades de se comunicarem com os outros membros, estando muitas vezes desinformados sobre as decisões tomadas em grupo. Ocorreram casos de alterações sem aviso prévio e sem a aprovação da equipe, o que fazia com que muitas dessas alterações fossem revogadas, por não estar de acordo com o planejado.

Houve também situações em que membros precisaram sair do projeto, devido a ineficiência no trabalho, que ocasionou atrasos, descontentamento do cliente, redução de escopo do projeto e replanejamento das tarefas e prazos. Além disso, houve dificuldades em manter alguns membros da equipe motivados, em razão de todos os atrasos e obstáculos. Um dos maiores fatores que desmotivava alguns participantes do projeto, que foram contemplados com uma bolsa, diz respeito ao atraso de pagamento, o que provocou insatisfação e resistência ao trabalho.

8 RESULTADOS

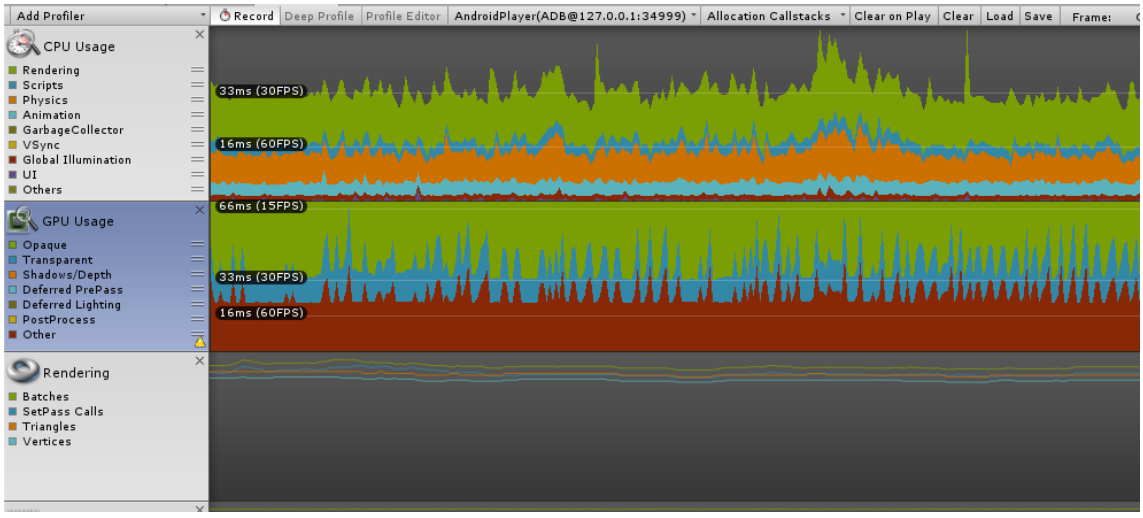
Os resultados apresentados nesta seção foram obtidos a partir de uma *build* de desenvolvimento do jogo, o que permite coletar dados de *performance* durante o tempo de execução, através do *Profiler* da *Unity*.

É importante salientar que a utilização da inspeção de desempenho ocasiona um pequeno comprometimento na *performance* geral da aplicação, uma vez que há um custo adicional para coletar e enviar os dados para o *Profiler*. Dito isso, os resultados obtidos não correspondem (com precisão) o desempenho real da aplicação sendo executada em um determinado dispositivo.

8.1 Resultados obtidos com o Profiler

O *Galaxy J6*, celular de categoria intermediária (*mid end*), foi o dispositivo utilizado para analisar o desempenho geral do jogo. A figura 65 apresenta os gráficos obtidos pelo *Profiler* da *Unity*. Com isso, é possível afirmar que o celular em questão está sendo limitado pela sua *GPU*, sendo categorizado como "*GPU Bound*". Além disso, a principal ocupação da *CPU* ocorre no processo de *rendering*, enquanto que na *GPU*, a renderização de objetos opacos está consumindo mais desse componente.

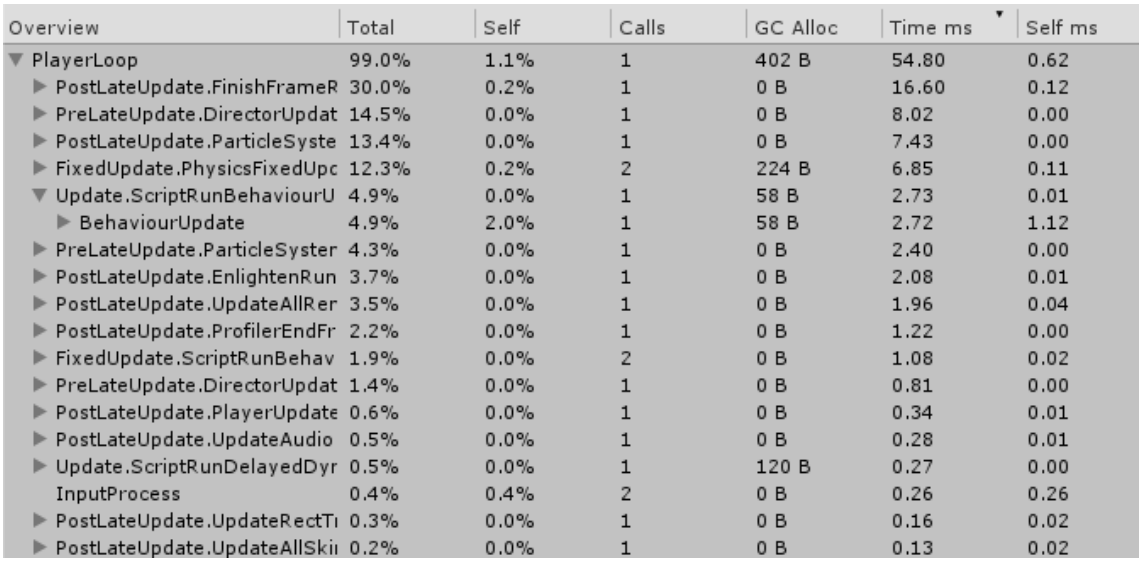
Figura 65 – Gráficos de desempenho no *Galaxy J6*



Fonte: Concebido pelo autor

A figura 66 apresenta uma lista dos processos que estão sendo executados pela *CPU*, sendo ordenados pelo seu tempo de execução (em milissegundos). É possível identificar que a finalização de quadro está ocupando 30% do total de processamento, levando cerca de 16,6 milissegundos para ser processado. Enquanto que o sistema de física está ocasionando o maior número de alocações dinâmicas (verificar aba "*GC Alloc*"). Por outro lado, os processos que correspondem aos *scripts* em execução ocupam apenas 4,9% do processamento total e realiza uma pequena quantia de alocação dinâmica (58 *Bytes*).

Figura 66 – Lista de processos durante o tempo de execução do jogo



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	99.0%	1.1%	1	402 B	54.80	0.62
PostLateUpdate.FinishFrameR	30.0%	0.2%	1	0 B	16.60	0.12
PreLateUpdate.DirectorUpdat	14.5%	0.0%	1	0 B	8.02	0.00
PostLateUpdate.ParticleSyste	13.4%	0.0%	1	0 B	7.43	0.00
FixedUpdate.PhysicsFixedUpc	12.3%	0.2%	2	224 B	6.85	0.11
Update.ScriptRunBehaviourU	4.9%	0.0%	1	58 B	2.73	0.01
BehaviourUpdate	4.9%	2.0%	1	58 B	2.72	1.12
PreLateUpdate.ParticleSyster	4.3%	0.0%	1	0 B	2.40	0.00
PostLateUpdate.EnlightenRun	3.7%	0.0%	1	0 B	2.08	0.01
PostLateUpdate.UpdateAllRer	3.5%	0.0%	1	0 B	1.96	0.04
PostLateUpdate.ProfilerEndFr	2.2%	0.0%	1	0 B	1.22	0.00
FixedUpdate.ScriptRunBehav	1.9%	0.0%	2	0 B	1.08	0.02
PreLateUpdate.DirectorUpdat	1.4%	0.0%	1	0 B	0.81	0.00
PostLateUpdate.PlayerUpdate	0.6%	0.0%	1	0 B	0.34	0.01
PostLateUpdate.UpdateAudio	0.5%	0.0%	1	0 B	0.28	0.01
Update.ScriptRunDelayedDyr	0.5%	0.0%	1	120 B	0.27	0.00
InputProcess	0.4%	0.4%	2	0 B	0.26	0.26
PostLateUpdate.UpdateRectTı	0.3%	0.0%	1	0 B	0.16	0.02
PostLateUpdate.UpdateAllSkii	0.2%	0.0%	1	0 B	0.13	0.02

Fonte: Concebido pelo autor

A partir disso, é possível perceber que há a possibilidade de realizar melhorias na otimização das lógicas e processos que envolvam o sistema de física da *Unity*, além de necessitar encontrar meios que reduzam a sobrecarga da *GPU*, mais especificamente para esse celular.

8.1.1 Culling frustum e static batch

A tabela 1 demonstra os resultados obtidos no desempenho utilizando os mecanismos de *Occlusion Culling* e agrupamento estático. Os valores apresentados no resultado consistem em: taxa de quadros por segundos (*FPS*); quantidade de *Draw Calls* estáticos; quantidade de *Draw Calls* dinâmicos; número de conjuntos estáticos e dinâmicos.

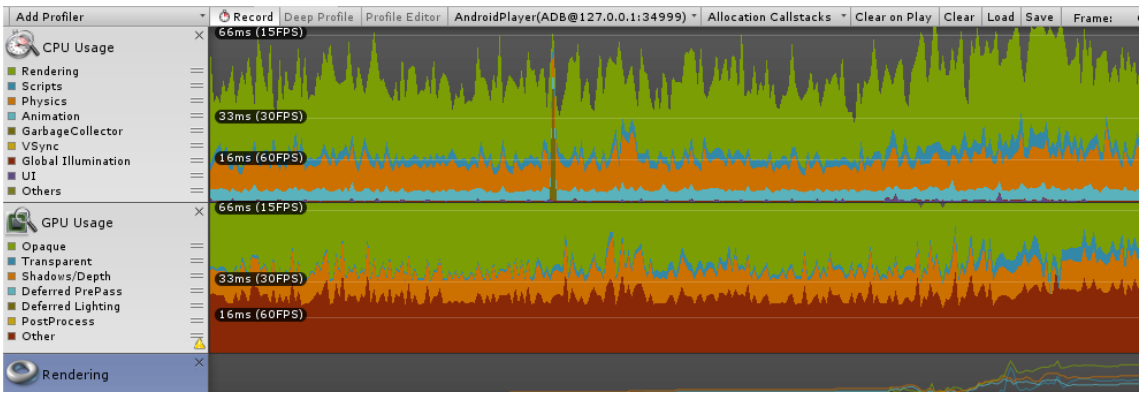
Tabela 1 – Comparativo de desempenho com *Occlusion Culling* e agrupamento estático

Occlusion Culling e objetos estáticos	FPS	Draw Calls (Estáticos)	Draw Calls (Dinâmicos)	Batch Dinâmico	Batch Estático
Presente	21	34	477	14	84
Ausente	12	322	0	39	0

Fonte: Concebido pelo autor

A figura 67 apresenta os gráficos do *Profiler* sem a utilização de agrupamento estático e *Occlusion Culling*. Pode-se perceber (comparando com a figura 65) que há um uso mais intenso do sistema de forma geral, quando não se utiliza esses mecanismos.

Figura 67 – Gráficos de desempenho sem *Occlusion Culling* e *Static Batching*



Fonte: Concebido pelo autor

8.1.2 Simplificação de texturas

Nesta seção será apresentado (na tabela 2) os ganhos obtidos no uso de memória utilizando os mecanismos de otimização de texturas, descritos na seção 6.2. Para efeitos de comparação, são utilizadas duas variantes do projeto: uma sem realizar compressão nos mapas de texturas; e outra realizando o caminho inverso da otimização (mal otimizada), com a utilização de texturas grandes, habilitando a propriedade *Read/Write* e utilizando *mipmaps*.

Tabela 2 – Comparativo do uso de memória entre as diferentes estratégias de otimização

Uso de texturas	FPS	Uso de memória – Apenas as texturas	Uso total de memória – Todos os assets
Otimizado	21	138.1 mb	207.7 mb
Sem compressão	15	190.7 mb	260.3 mb
Mal otimizado	15	279.6 mb	349.5 mb

Fonte: Concebido pelo autor

Com os dados obtidos, verifica-se que não utilizar as ferramentas de compressão de texturas acarretou um aumento do consumo de memória de cerca de 50 *megabytes* (*mb*). Enquanto que a variante mal otimizada obteve aumento significativo no uso de memória, de 207,7 *mb* para 349,5 *mb*, correspondendo a um aumento de 141,8 *mb*, o que é bastante para um dispositivo móvel. Além disso, houve uma queda considerável na taxa de quadros por segundo, correspondendo a uma perda média de 7 quadros por segundo.

8.1.3 Ajustes de iluminação

Como descrito na seção 6.6, alguns processos de otimização de iluminação foram realizados no projeto, sendo apresentado nesta seção os ganhos obtidos para os 3 cenários do jogo.

8.1.3.1 Otimizações na cidade

O cenário da cidade obteve um pequeno ganho de *performance*, embora o processo de renderização tenha sido otimizado, devido a redução da quantidade de *Draw Calls*. Acredita-se que os ganhos foram limitados pelo uso do *Profiler* e pela limitação da *GPU* do aparelho utilizado para teste, podendo ser um ganho mais significativo para aparelhos com melhor processador gráfico. A tabela 3 apresenta os resultados obtidos.

Tabela 3 - Comparativo entre otimizações no cenário da cidade

Otimização de iluminação	**FPS**	***Draw Calls***	**Total de *batches***
Antes	20	420	147
Depois	21	284	123

Fonte: Concebido pelo autor

8.1.3.2 Otimizações na casa da tia

O cenário da casa da tia apresentou um ganho de desempenho consideravelmente alto, com média de 6 quadros (aproximadamente) por segundo adicionais, tornando esse cenário mais jogável. Algumas das otimizações realizadas neste cenário correspondem a: limitar o tamanho dos *light maps*; redução das amostras calculadas e da qualidade geral dos parâmetros dos *light maps*; além da diminuição do número de fontes de luz, utilizando apenas uma fonte de luz direcional. Os resultados obtidos podem ser vistos na tabela 4.

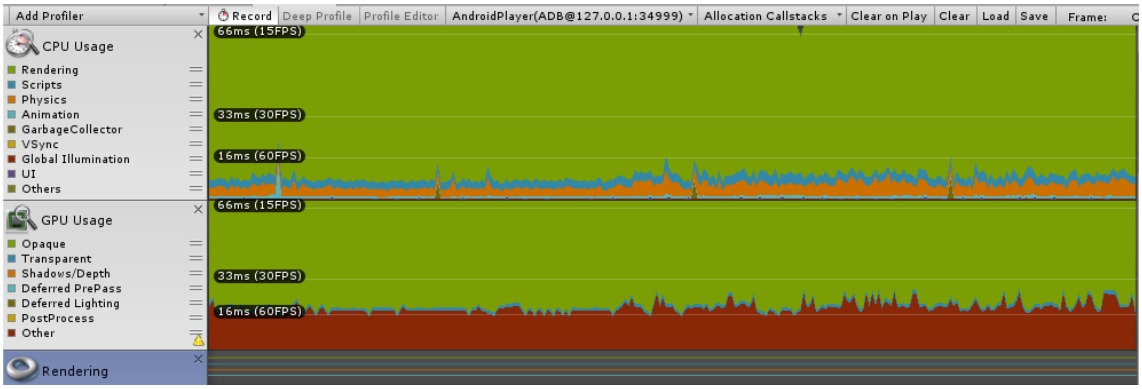
Tabela 4 - Comparativo entre otimizações no cenário da casa da tia

Otimização de iluminação	**FPS**	***Draw Calls***	**Total de *batches***
Antes	11	671	500
Depois	17	230	178

Fonte: Concebido pelo autor

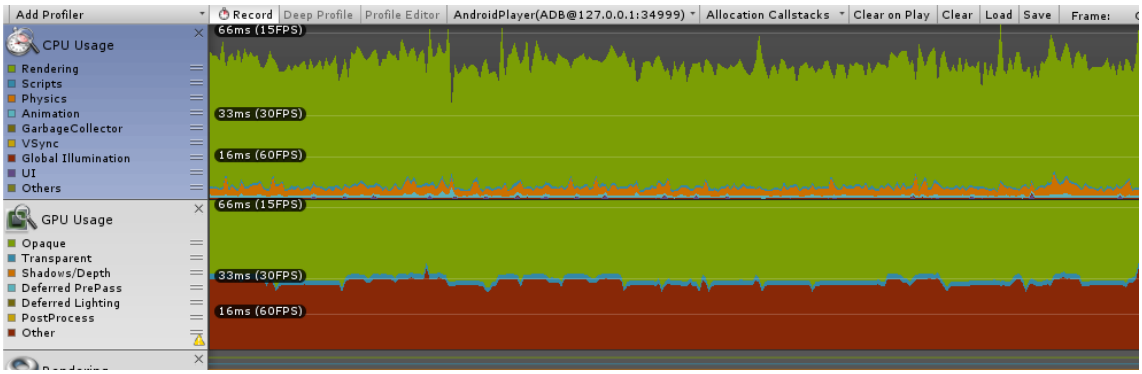
Para complementar, as figuras 68 e 69 apresentam os gráficos obtidos pelo *Profiler* antes e depois da otimização, respectivamente.

Figura 68 – Gráfico de desempenho sem otimização (casa da tia)



Fonte: Concebido pelo autor

Figura 69 – Gráfico de desempenho com otimização (casa da tia)



Fonte: Concebido pelo autor

8.1.3.3 No hospital

O cenário do hospital demonstrou ser o ambiente que mais necessitava de otimização, chegando a ter um ganho médio (aproximado) de 12 *FPS*, após da realização da otimização, utilizando uma estratégia semelhante a aplicada ao cenário da casa da tia. A tabela 5 apresenta os resultados obtidos para este cenário.

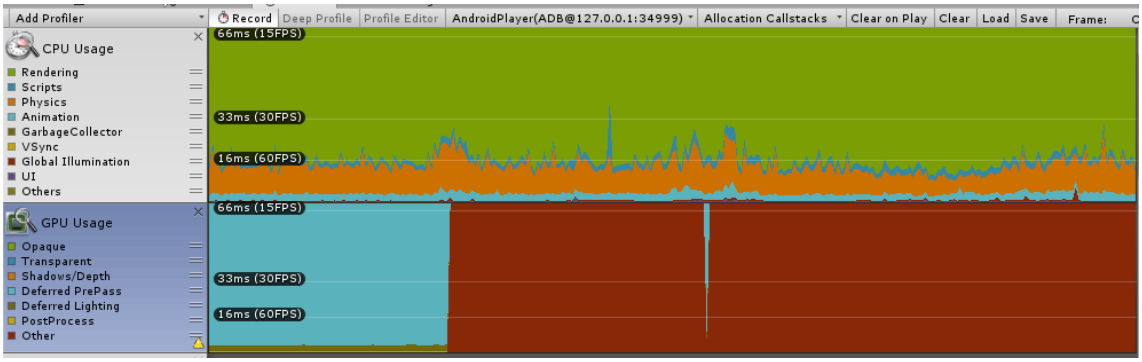
Tabela 5 - Comparativo entre otimizações no cenário do hospital

Otimização de iluminação	FPS	*Draw Calls*	Total de *batches*
Antes	4	1526	1338
Depois	16	391	155

Fonte: Concebido pelo autor

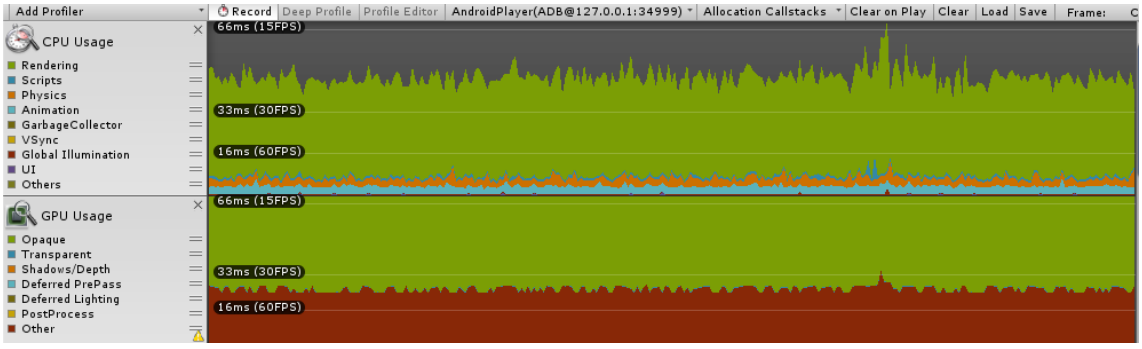
Após a realização da otimização, nota-se que houve uma grande redução na quantidade de *draw calls* e no número de *batches* de objetos, o que garante uma melhora considerável no processo de renderização da cena. As figuras 70 e 71 apresentam os gráficos obtidos pelo *Profiler*, antes e depois de realizar a otimização, respectivamente.

Figura 70 - Gráfico de desempenho sem otimização (hospital)



Fonte: Concebido pelo autor

Figura 71 - Gráfico de desempenho com otimização (hospital)



Fonte: Concebido pelo autor

8.2 Resultados da aplicação final

Por fim, a tabela 6 apresenta o desempenho da *build* final do jogo em diversos aparelhos, indo de categoria de baixo custo (*low end*) até os topo de linha (*high end*).

Tabela 6 – Comparativo final do desempenho entre vários dispositivos

Dispositivo	**Ano de lançamento**	**Categoria**	**FPS médio – Qualidade Baixa**	**FPS médio – Qualidade Alta**
Galaxy J6	2018	Mid-end	30	25
Galaxy S8	2017	High-end	30	59
Galaxy Tab A	2017	Low-end	27	18
Moto E (2ª Geração)	2015	Low-end	20	17
Moto X (2ª Geração)	2014	Mid-end	25	26
Zenfone 4 Max	2017	Mid-end	30	23
Lenovo Zuk Z2 Pro	2016	High-end	30	51
LG K10	2016	Low-end	30	24

Fonte: Concebido pelo autor

É possível notar que o desempenho médio para os dispositivos intermediários está na faixa de 24 *FPS* na qualidade alta e 30 *FPS* na qualidade baixa. Ao mesmo tempo em que os dispositivos topo de linha conseguem alcançar taxa de quadros maiores do que 30 *FPS*, correspondendo a um ótimo resultado.

Por outro lado, os dispositivos de menor qualidade apresentaram uma taxa média de 25 *FPS* na qualidade baixa, representando uma taxa aceitável e jogável, enquanto que na qualidade alta ficaram por volta de 20 *FPS*, não sendo um resultado ideal, porém ainda pode ser jogável.

8.3 Resultados do playstest

Durante o processo de desenvolvimento do projeto, houve algumas sessões *playtest* com versões *alfas* do projeto para um pequeno grupo de jogadores, para avaliarem a qualidade de algumas características do jogo. Embora a versão final do produto seja muito distinta da versão utilizada nos *playtests*, ainda é válido apresentar os resultados obtidos no Apêndice B, pois alguns dos *feedbacks* recebidos foram utilizados para levantar requisitos que seriam posteriormente implementados na versão final do produto.

9 CONCLUSÕES E TRABALHOS FUTUROS

De acordo com o que foi apresentado neste trabalho, pode-se perceber que o processo de desenvolvimento de um jogo exige muito trabalho, planejamento e tomadas de decisão para se obter um bom resultado. Demonstrando a necessidade de realizar vários testes e construção de protótipos para validar as ideias e propostas do jogo. Além de apresentar a importância de analisar o desempenho da aplicação na plataforma alvo e as práticas de otimização as quais devem ser adotadas para se obter uma melhor *performance* no jogo, melhorando a experiência e satisfação do jogador.

Devido ao fato do *game* ainda não estar disponível na *Play Store*, para o público poder baixar e jogar, até o momento da escrita deste documento, não foi possível analisar e incluir as avaliações de usuários do produto final neste trabalho. Embora ainda possa ser abordado e avaliado em trabalhos futuros.

Mesmo com a produção do jogo finalizada, é possível destacar, a partir dos resultados apresentados no capítulo 8, que ainda há espaço para melhorar (ainda mais) o desempenho do jogo, utilizando técnicas e metodologias que antes não eram conhecidas pela equipe de desenvolvimento do projeto. Devido ao fato (principalmente) de que algumas delas foram disponibilizadas muito recentemente, como o *Entity Component System* (ECS) da *Unity*, que foi disponibilizado a partir da versão 2019.1, lançado no dia 16 de abril de 2019. Estas técnicas e metodologias serão descritas brevemente nos seguintes tópicos deste capítulo.

9.1 Scriptable Objects

Os objetos “scriptáveis” (*Scriptable Objects*) são classes úteis para armazenar dados e estados compartilhados, que podem ser acessados por vários objetos e *scripts*, além de centralizar e modularizar esses dados, facilitando a alteração e manipulação deles, tanto pelos programadores quanto outros desenvolvedores com pouco (ou nenhum) conhecimento de programação, como artistas e *designers*.

De acordo com Fine (2016), muitos projetos desenvolvidos na *Unity* são implementados utilizando apenas classes que herdam da *MonoBehaviour*, podendo se deparar com diversos problemas como: perda dos valores de dados ao carregar o jogo; aumento da granularidade dos arquivos de cenas e *Prefabs*, dificultando projetos realizados em equipe (riscos de conflito); além da classe *Monobehaviour* possuir várias funções *callbacks*, provocando dificuldades de depuração do código e identificação de erro.

Uma alternativa indicada por Fine (2016), consiste no uso de objetos scriptáveis, segundo ele, estas classes:

- Não podem ser associadas a um *GameObject* ou *Prefab*;
- Não podem ser criadas na cena do jogo;
- Podem ser serializadas e manipuladas no editor da *Unity*;
- Possuem apenas 3 funções *callback*: *OnEnable*, *OnDisable* e *OnDestroy*;
- Permitem o desenvolvimento de métodos próprios.

Deste modo, *Scriptable Objects* são mais simples e fáceis de manter do que *Monobehaviours*. Além disso, outros benefícios de sua utilização foram apontados por Fine (2016), são eles:

- Melhor desacoplamento da lógica e dos dados;
- Suporte a ferramentas que transcrevem dados de arquivos JSON;
- Controle sobre a granularidade dos arquivos;
- Possibilidade de ser utilizado como uma base de dados real;
- Melhora a estrutura da arquitetura do projeto;
- Favorece o uso de padrões de projetos, como *Singleton* e *Strategy*;

9.2 Entity Component System

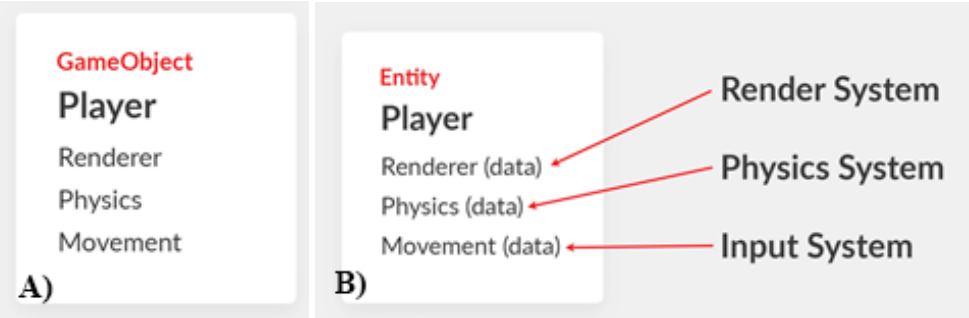
Como foi visto no capítulo 6, é necessário adotar diversas estratégias e técnicas de otimização no desenvolvimento tradicional de aplicações na *Unity*. Com o intuito de mudar o foco dos desenvolvedores em apenas implementar o jogo, sem precisar se preocupar com otimização, a *Unity Technologies* vem reconstruindo os fundamentos do desenvolvimento de aplicações na *engine*, focando em um novo paradigma de alta performance em *multithread*, denominado *Data-Oriented Technology Stack* (DOTS), que possui como lema "*Performance* por padrão" e inclui os seguintes recursos:

- *Entity Component System* (ECS): Permite escrever códigos de alta performance;
- *C# Job System*: Permite executar códigos *C#* em *multithread*, de forma eficiente e segura;
- *Burst Compiler*: Traduz códigos em *C#* para código nativo altamente otimizado;

De acordo com o canal Brackeys (2018)[22], *ECS* consiste em uma nova forma de escrever códigos na *Unity*, mudando do paradigma tradicional, orientado aos *GameObjects*, para o paradigma orientado aos dados. Até o momento, toda a lógica escrita na *Unity* era centralizada em *GameObjects* e na classe *Monobehaviour*. Já em ECS, a abordagem consiste em dividir este objeto em 3 partes: Entidades (*Entities*), responsáveis por agrupar componentes, semelhante aos objetos tradicionais (*GameObject*), porém são mais leves; Componentes (*Components*) contêm apenas os dados, sem incluir nenhuma lógica; Sistemas (*Systems*) são responsáveis por conter as lógicas e comportamentos, utilizando os dados presentes nos componentes de uma entidade. A figura 72(a) apresenta um diagrama de um modelo de objeto tradicional na *Unity*, e a figura 72(b) demonstra um modelo correspondente no padrão *ECS*.

[22] Disponível em: https://www.youtube.com/watch?v=_U9wRgQyy6s&t=191s

Figura 72 – Comparativo entre objetos tradicionais e ECS



Fonte: canal do *YouTube* "Brackeys"

No paradigma tradicional, o objeto "*Player*" possui 3 componentes: "*Renderer*", "*Physic*" e "*Movement*". Em que cada um contêm dados e lógicas que determinam o comportamento deste objeto. Já no novo paradigma (Orientado a dados), o "*Player*" corresponde a uma entidade, que não possui nenhum dado e lógica, podendo ser encarado como um índice; o "*Renderer*", "*Physics*" e "*Movement*" correspondem aos componentes, que possuem apenas dados, como velocidade, massa e etc; e, por fim, o "*Render System*", "*Physics System*" e o "*Input System*" representam os sistemas, que possuem a lógica e utilizam os dados de determinados componentes para realizar algum comportamento (representado pelas setas na figura 50(a)).

Os benefícios de utilizar o padrão *ECS* da *Unity*, apontados pelo canal Brackeys (2018), estão em fornecer uma melhor forma de estruturar os dados e escrever códigos que são:

- Extremamente otimizados;
- Simples e compreensíveis;
- Reutilizáveis;
- Compilados pelo *Burst Compiler*;
- Integrados com o *C# Job System*;

Permitindo, desta forma, fazer melhor uso dos múltiplos núcleos dos processadores modernos. Para concluir, a *Unity* forneceu no seu *Gitub* uma tirinha, ilustrando o funcionamento do sistema ECS, que é apresentada no anexo A.

REFERÊNCIAS

AZEVEDO, Eduardo. **Teoria da Computação Gráfica**. Rio de Janeiro: Editora Campus, Ltda., 2003.

FLEURY, A.; NAKANO, D.; CORDEIRO, J. **Mapeamento da Indústria Brasileira e Global de Jogos Digitais**. São Paulo: NPGT / Escola Politécnica / USP, 2014.

GONZALEZ, Jonathan. "**Maximizing Your Unity Game's Performance**". 2017. Disponível em: < https://cgcookie.com/articles/maximizing-your-unity-games-performance. Acessado em: 07/04/2019 >. Acesso em: 08/04/2019.

_____, "**HISTÓRIA DO TCE-PB**". [20-?]. Disponível em: <http://tce.pb.gov.br/institucional/breve-historia>. Acesso em: 20/04/2019.

HUGHES, J. F. et al. **Computer Graphics: principle and practice**. 3.ed. New Jersey: Pearson Education, Inc., 2014.

KING, D.; GRIFFITHS, M.; DELFABBRO, P. Video Game Structural Characteristics: A New Psychological Taxonomy. **International Journal of Mental Health and Addiction**. 90-106, nov. 2009.

LAVID. "**De laboratório a núcleo**, conheça a história do Lavid". [20-]. Disponível em: <http://lavid.ufpb.br/index.php/quem-somos/>. Acesso em: 20/04/2019.

MATT. "**Unity3D Mesh Collider vs. Box Collider**". 2016. Disponível em: <http://blog.roguecode.co.za/Unity3D-Mesh-Collider-vs-Box-Collider>. Acesso em: 06/04/2019.

MEIRELLES, Fernando. 29ª Pesquisa Anual do Uso de TI. 2018. Disponível em: < https://eaesp.fgv.br/sites/eaesp.fgv.br/files/pesti2018gvciappt.pdf>. Acesso em: 01/05/2019.

NEW way of CODING in Unity! ECS Tutorial. Direção: Brackeys. Dinamarca, 2018. 9'09''. Disponível em: <https://www.youtube.com/watch?v=_U9wRgQyy6s> Acesso em: 15/04/2019.

_____, ____. O crescimento da indústria de games no Brasil. **Dino**. 10 ago. 2018. Negócio. Disponível em: <https://exame.abril.com.br/negocios/dino/o-crescimento-da-industria-de-games-no-brasil/>. Acesso em: 01/05/2019.

PACHECO, Paula. Mercado de games no Brasil cresce, apesar da crise. **Jornal Estado de Minas**. Minas Gerais. 24 jul. 2018. Economia. Disponível em: <https://www.em.com.br/app/noticia/economia/2018/07/24/internas_economia,975277/mercado-de-games-no-brasil-cresce-apesar-da-crise.shtml>. Acesso em: 01/05/2019.

PAYPAL BRASIL. “**82% dos brasileiros costumam jogar em smartphones”**. 2018. Disponível em: < https://www.paypal.com/stories/br/82-dos-brasileiros-costumam-jogar-em-smartphones>. Acesso em: 01/05/2019.

PORTER. “**Unity:** Now You’re Thinking with Components”. 2013. Disponível em: <https://gamedevelopment.tutsplus.com/articles/unity-now-youre-thinking-with-components--gamedev-12492>. Acessado em: 29/04/2019.

RITTERFELD, U.; CODY, M.; VORDERER, P. **Serious games: Mechanisms and Effects**. New York: Routledge, 2009.

ROSA, Natalie. Mercado de games vem crescendo em todo o Brasil, revela pesquisa. **Canaltech**. [S.l]. 29 jul. 2018. Games. Disponível em: < https://canaltech.com.br/games/mercado-de-games-vem-crescendo-em-todo-o-brasil-revela-pesquisa-116972/> Acesso em: 01/05/2019.

SATO, A. Game Design e Prototipagem: Conceitos e Aplicações ao Longo do Processo Projetual. **SBGames**, v.9, Florianópolis, 74-84, nov. 2010.

SCHELL, Jesse. **The Art of Game Design: A book of lenses.** Burlington: Elsevier, 2008.

TEIXEIRA, Nickolas. “**Stack vs Heap. Whta’s the difference and why should I care?**”. 2018. Dispinível em: <https://medium.com/@nickteixeira/stack-vs-heap-whats-the-difference-and-why-should-i-care-5abc78da1a88>. Acesso em: 04/05/2019.

THE ENTERTAINMENT SOFTWARE ASSOCIATION. **Essencial Facts about the computer and video game industry**: 2018 sales, demography and usage data.

UNITE 2015 – Overthrowing the Monobehaviour Tyranny in a Glorious Scriptable Object Revolution. Apresentação: Richard Fine. [S.l]: Unity, 2015. 57’32’’. Disponível em: < https://youtu.be/6vmRwLYWNRo >. Acesso em: 15/04/2019.

UNITE Europe 2015 – How we optimized our mobile game – Project Monsters. Apresentação: Phil Lira. Amsterdã: Unity, 2015. 32’03’’. Disponível em: <https://youtu.be/G0IdA_CHCUs> Acesso em: 05/04/2019.

UNITE Europe 2016 – Optimizing Mobile Applications. Apresentação: Ian Dundore e Mark Harkness. Amsterdã: Unity, 2016. 45’49’’. Disponível em: <https://youtu.be/j4YAY36xjwE> Acesso em: 03/04/2019.

UNITE Europe 2017 – Performance optimization for beginners. Apresentação: Kerry Turner e Matt Schell. [S.l]: Unity, 2017. 27’36’’. Disponível em: <https://www.youtube.com/watch?v=1e5WY2qf600&feature=youtu.be> Acesso em: 02/04/2019.

UNITE Europe 2017 – Squeezing Unity: Tips for raising performance. Apresentação: Ian Dundudore. Material: Ian Dundore e Mark Harkness. [S.l]: Unity, 2017. 50’06’’. Disponível em: <https://youtu.be/_wxitgdx-UI> Acesso em: 03/04/2019.

UNITY. “**Fundamentals of Unity UI**”. [20-]. Disponível em: <https://unity3d.com/learn/tutorials/topics/best-practices/fundamentals-unity-ui>. Acesso em: 22/04/2019.

UNITY. “**Performance by Default**”. 2019. Disponível em: <https://unity.com/dots> Acesso em: 02/05/2019.

UNITY. “**The Profiler Window**”. [2016?]. Disponível em: <https://unity3d.com/pt/learn/tutorials/temas/performance-optimization/profiler-window?playlist=44069>. Acesso em: 25/04/2019

UNITY. “**The world’s leading real-time creation plataform**”. [20-]. Disponível em: <https://unity3d.com/unity?_ga=2.37825251.114713944.1557266174-27483104.1543878880>. Acesso em: 06/04/2019.

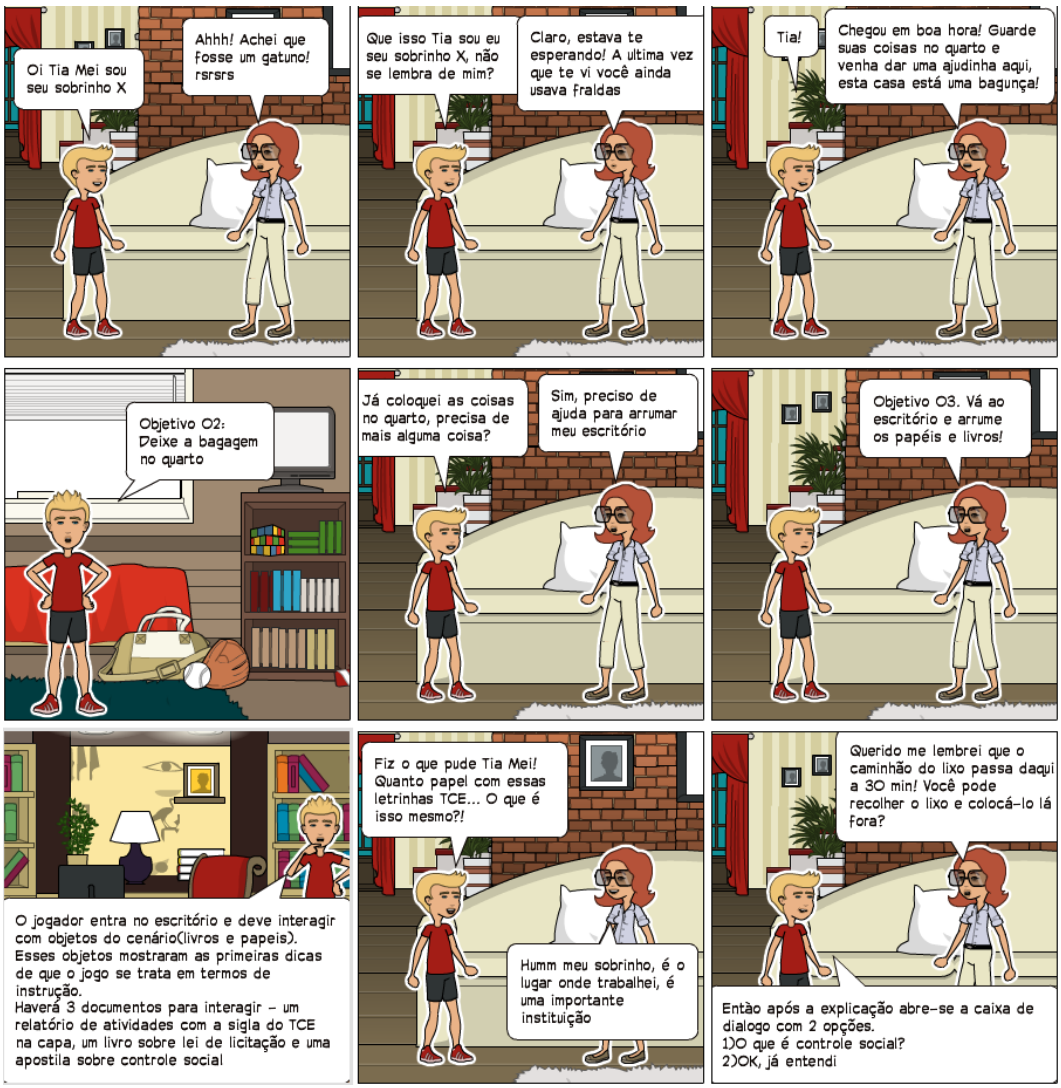
UNITY DOCUMENTATION. **Types of light**. V. 2019.1. Disponível em: <https://docs.unity3d.com/Manual/Lighting.html>. Acesso em: 20/04/2019.

UNITY MANUAL. **GameObject**. V. 2019.1. Disponível em: <https://docs.unity3d.com/Manual//class-GameObject.html>. Acesso em: 25/04/2019.

VINCE. “**The Many Different Types of Video Games & their Subgenres**”. 2018. Disponível em: <https://www.idtech.com/blog/different-types-of-video-game-genres. Acessado em: 06/05/2019>. Acesso em: 02/05/2019.

APÊNDICE A – PROTÓTIPO DE DIÁLOGOS DA NARRATIVA

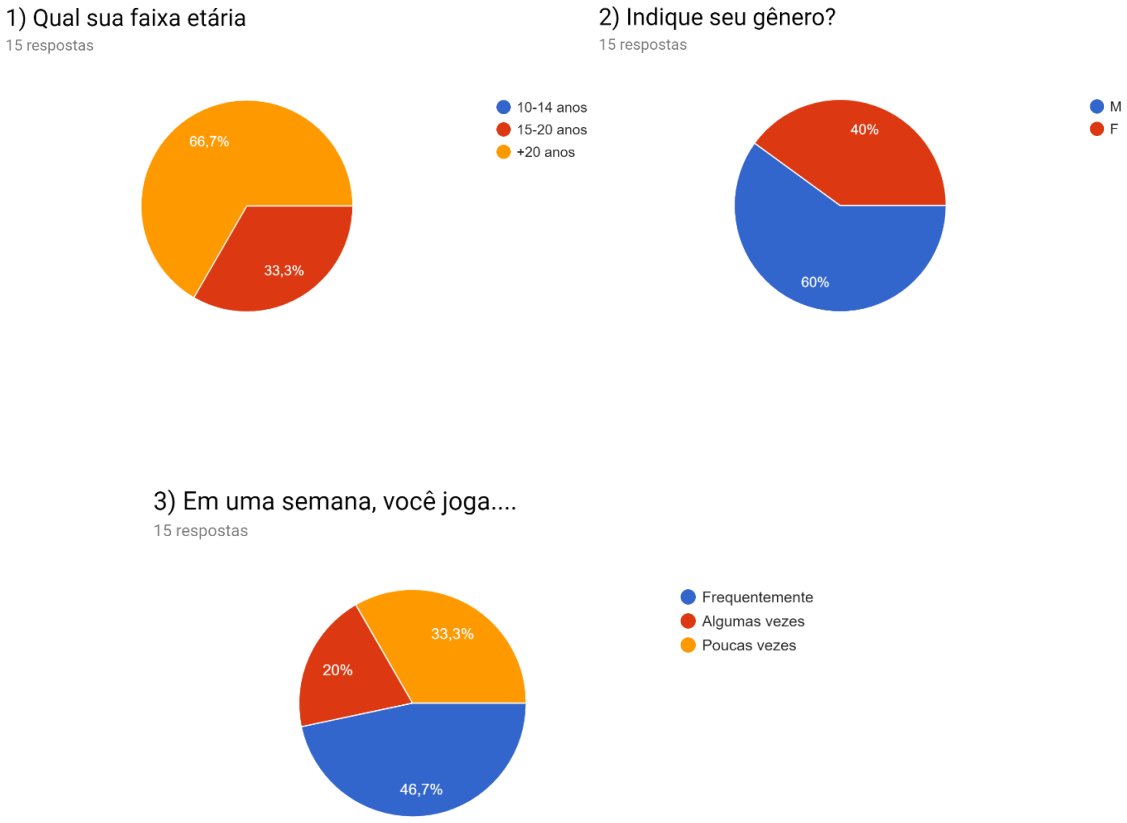
Figura 73 – Esboço da primeira missão



Fonte: Documento de *Design* do "Recruta Social"

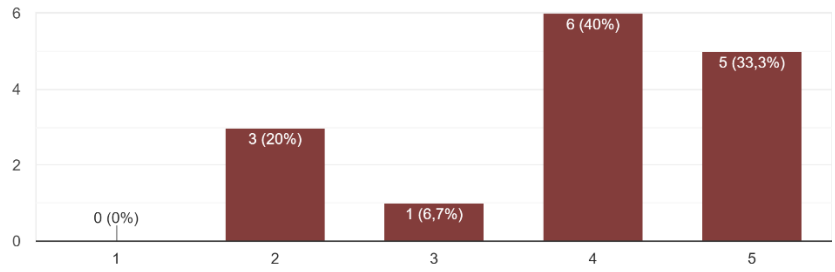
APÊNDICE B – AVALIAÇÕES DE PLAYTEST

Figura 74 – Avaliações de usuário realizada durante um *playtest* de um alfa



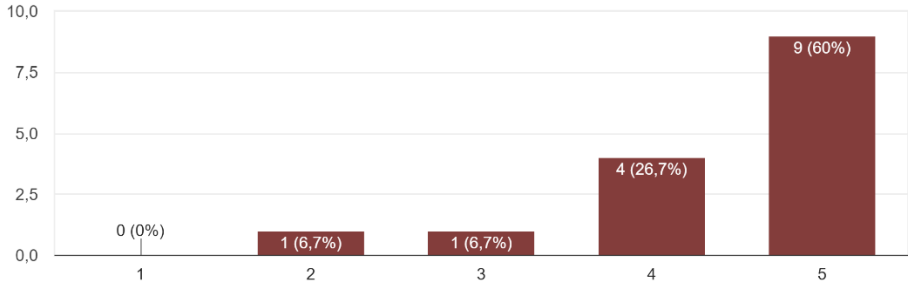
6) Dê uma nota de 1 a 5 para a HISTÓRIA DE ENTRADA. Considere 1 para a nota mais baixa e 5 para a mais alta.

15 respostas



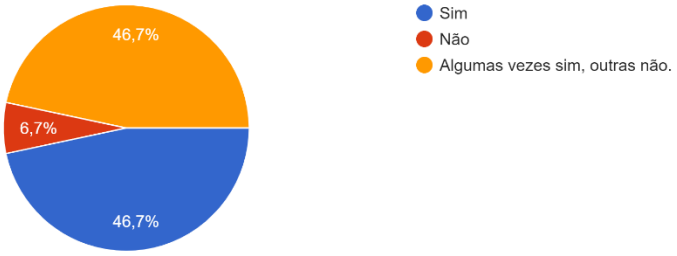
8) Dê uma nota de 1 a 5 para a INTERFACE e CENÁRIOS. Considere 1 para a nota mais baixa e 5 para a mais alta.

15 respostas



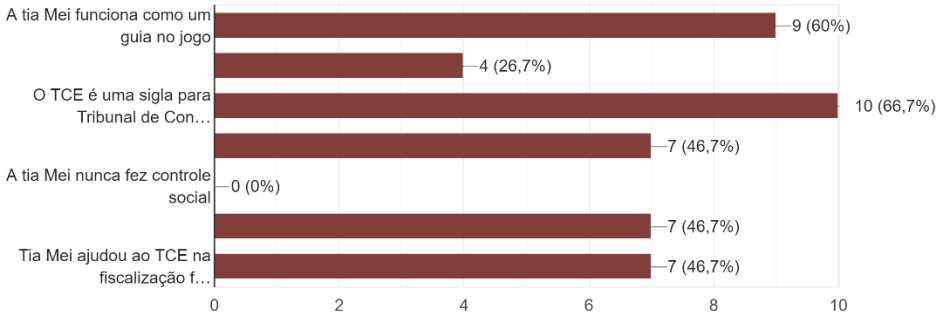
11) Você conseguiu identificar qual o objetivo a ser alcançado a cada momento no jogo?

15 respostas



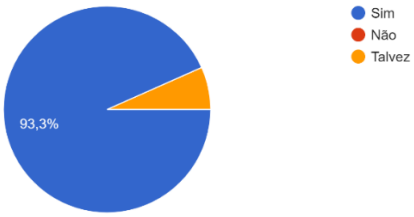
13) Indique dentre as alternativas abaixo o que foi possível entender no jogo

15 respostas



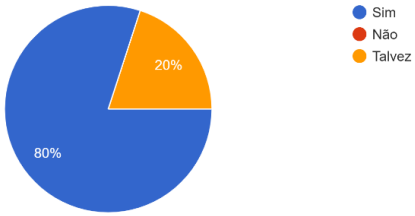
17) Você recomendaria o jogo para outra pessoa?

15 respostas



18) Você jogaria novamente?

15 respostas



Fonte: Questionário do *playtest* do jogo

ANEXO A – CONHECENDO ECS DA UNITY

Figura 75 – Introdução ao ECS (Continua)

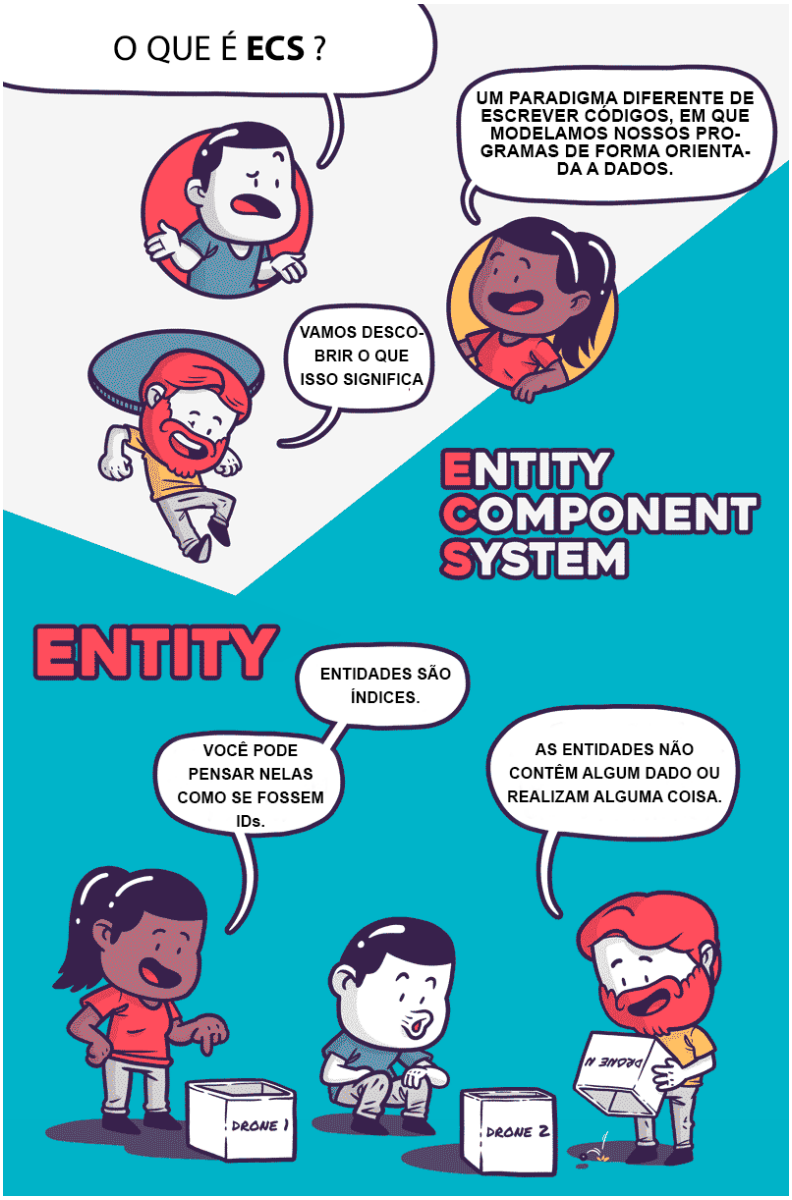
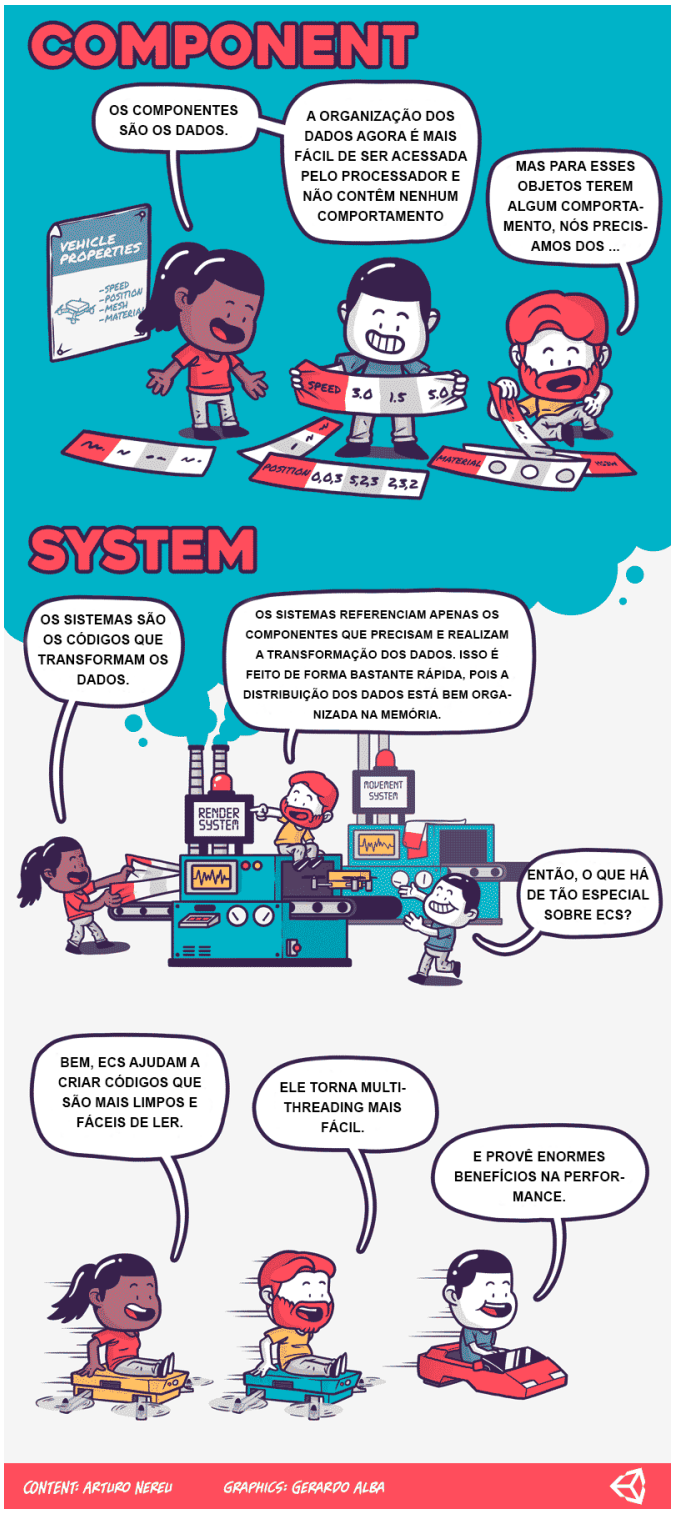


Figura 76 – Introdução ao ECS (Conclusão)



Fonte: Github da *Unity*[23] (tradução nossa)

[23] Disponível em: https://github.com/Unity-Technologies/EntityComponentSystemSamples/blob/master/Documentation%7E/getting_started.md